Towards Software Architecture Physiology: Identifying Vital Components[1]

Ilham Alloui, Sorana Cîmpan, Herve Verjus
University of Savoie – Polytech' Savoie - LISTIC Lab
B.P. 80439 - 74944 Annecy-le-Vieux Cedex - France
{ilham.alloui, sorana.cimpan, herve.verjus}@univ-savoie.fr

Abstract

Several architecture analysis methods are proposed in the literature for evaluating both the structure and the behavior of architectures. A parallel between humans and software systems leads to some interesting consideration on kinds of analysis that can be performed on a system architecture, such as the identification of vital element. Such identification improves the system architecture understanding and allows us to estimate and to estimate to what extent, a change on some components could impact the rest of the architecture.

1 Introduction

Like human, software system architecture has an anatomy with organs that fulfill well defined objectives. However while human's anatomy basically comprises the same components from a person to another with some variants (male, female), and with rather a well understood physiology (some of the organs are vital, others are not), in case of software, every system has its own architecture (which may be derived from a reference architecture, be part of a product family or follow an architectural style) and vital components are not always known a priori.

Many analysis approaches tackle either the structure or the behavior of software architecture [6] [7] during a specification process of a new system, but they generally do not help users identify architecture vital components during maintenance or evolution.

Using the parallel with humans, we think architecture analysis should also focus on the system physiology. This includes the identification of vital elements. Is there elements whose failure leads to the system degradation and even to its death? Furthermore, as it is not the same removing the heart or a finger, several situations can be identified for a software architecture: degraded functioning, blocking, etc.

Therefore one main issue is how to help software stakeholders identify vital components and have a view of the architecture physiology. Such identification enables both to enhance the system architecture and make it robust and to estimate to what extent, a change or failure on some components could impact the rest of the architecture.

We consider that this problem can be alleviated by the use of new methods for analyzing the system architecture. In particular we think about combining system dependency analysis methods with metrics.

We consider here a software system architecture as a set of services provided to the users to fulfill given objectives. Each service delivery may require one or many components and each component may be involved in one or many service delivery. A component encompasses part of the system functionality/data.

In the following we further detail the notion of vital component, which can be considered at different levels. We then provide a general algorithm for identifying vital components for a service (section 2). The algorithm is independent from the ADL used for representing the architectures, but uses functions which are formalism dependent. An implementation of these functions is introduced for ADLs based on process algebras, namely π-calculus (section 3). Section 4 provides some analysis based on vital components. Related work and discussions conclude the paper.

[1] In this paper by component, we mean any architectural element.

We gratefully acknowledge the financial support of the French ANR (National Research Agency) for the project "COOK: Réarchitecturisation des applications industrielles objets" (JC05 42872) in the framework of which this work is being developed.

2 Identifying vital components

As ideally a software system architecture is cohesive, globally all components are involved in providing the set of services and all could be consequently considered as vital. So rather than considering all services or some of them at once, we propose to identify first, vital components per service. We make the assumption that services are known and that we do not aim at discovering them. A consequence of that is if a component is vital for all services provided by the system architecture, it is considered as vital for the application (omnipresent component in vital components sets). Also, for a given component, we can identify its vital components, independently from a particular service. Thus we can identify three levels of vitality: service, component and system. In this paper we focus on service vital components.

Starting from one service contained in an architecture description, we can track backward to its potential providers by analyzing the supplier-consumer dependencies among components.

Let us consider *vital(S)* the set of vital components for a service *S*, and *vital(C,S)* the set of a component *C* vital components with respect to a service *S*. This set may contain additional information (cf. section 3).

This *vital(S)* set is initialized with a component which is tagged as the service provider[2]. It is incrementally computed (making use of *vital(C,S)*), as described in the algorithm given below.

$vital_0(S) = C$;
i=1 ;
do{
$vital_i(S) = vital_{i-1}(S) \cup$
$\{C_k \mid \exists j : C_j \in vital_{i-1}(S) \wedge C_k \in vital(C_j, S)\}$
} while $vital_i(S) != vital_{i-1}(S)$
$vital(S) = vital_i(S)$

Computing the set *vital(C,S)* depends on the language used for the architecture description. In the case of ADLs that take into account behavior, the computing is made by parsing the component's behavior description. Other cases may be considered, based on dependency analysis techniques [6].

Several ADLs take into account behavior by means of process algebras like π-calculus (ArchWare ADL [3][2], π-Space [4]) or CSP (Wright, DynamicWright [5]). We consider here the case of π-calculus based ADLs.

In the following section we show how for such ADLs, the set *vital(C,S)* is computed given a service *S* and a component *C*.

[2] We assume that such a tagging is possible.

3 The context of π-calculus based ADLs

A behavior expresses in scheduled way both the interaction of component and its internal computation. Behaviors can be connected through connections, along which values can be transmitted. The actions concern communication, *i.e.* sending and receiving information via connections, as well as internal computing. They are scheduled using π-calculus operators for expressing sequence, choice, composition and replication [3]. Architectural components are defined by composing behaviors, communicating through connections[3].

The *vital(C,S)* set is computed by parsing the component behavior, taking into account the actions and the operators that compose it. Internal actions are not considered. We focus on communication actions in order to identify the connected components and the dependency to these components for the given service. If in the behavior of *C* there is a receiving action of a value via a connection, then *C* is attached to another component that sends values on the same connection. Both behaviors are synchronized via this exchange.

The *vital(C,S)* set is initialized to the empty set when we start constructing it. At each action parsing and depending on the operator that precedes the action, new components are added to the set *vital(C,S).* Each element in the set is a triplet indicating:

- a component *Ci* which is vital for *C*,
- a connection of *Ci* on which the communication among *Ci* and *C* takes place
- the communication action of *Ci* (a send or a receive).

These three elements are needed in order to determine where to start *Ci*'s behavior parsing when constructing *vital(Ci,S).* Interactions that *Ci* behavior might contain but which are to be made after *Ci*'s interaction with *C* are not relevant for the service *S.*

In the following, we detail how the *vital(C,S)* set is modified when sequence operator is encountered. Similar treatments are made in the case of choice, composition and replication operators. Nevertheless, further aspects are to be considered, especially in the case of choice operators, where instead of adding only one vital component, a set of alternative components is added. These aspects are not addressed here..

The sequence operator among two actions *a1.a2* indicates that action *a1* takes place followed by action *a2*. The second action cannot take place until the first action is completed.

Let us thus consider that *cComputing* is the behavior of component *C*. As mentioned, the behavior is parsed

[3] The composition of such behaviors stands for synchronous concurrent processes.

from end to beginning; so in order to show how sequence is handled, let us consider that *cComputing* is *behaviourB.actionA, i.e., C* behaves as *behaviorB* and then performs *actionA.*

In the case where *actionA* is ***via*** *connection1* ***receive*** *value*, the triplet (*Cj*, *connection1*, *send*), where *Cj* is the component to which *connection1* is attached, is added to the *vital(C,S)* set. In the case where *actionA* is ***via*** *connection1* ***send*** *value* the triplet (*Cj*, *connection1*, *receive*), where *Cj* is the component to which *connection1* is attached, is added to the *vital(C,S)* set.

4 Further analysis based on vital components identification

Up till now we focused on how to find out components potentially required for a service. In this section, we go a step further in the analysis by considering the consequence on the system of vital components' failure. Indeed determining vital components for a service/component/system may have different effect from one part of the system to another. For instance if a service can potentially be accomplished two ways (*i.e.*, presence of a choice in the behavior), if a problem occurs on one way, there is always the possibility to follow the second way. In contrast, if a service is to be accomplished only one way (*i.e.*, presence of a sequence of actions only), a failure in the action path results in blocking the service. Based on those considerations, we propose to use metrics to decide whether a part of a system will be blocked, degraded or unaffected by a failure.

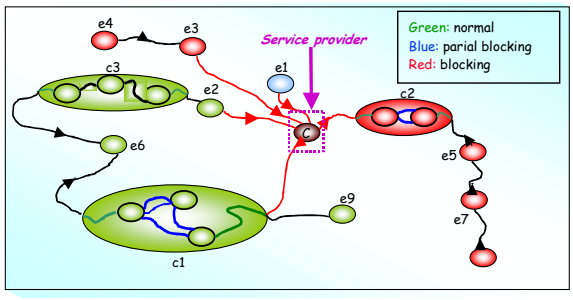

Figure 1 : Severity degree related graph

With this respect, we have experimented some algorithms based on a metric we named *change severity degree* to estimate the impact of a failure on the system. This metric is defined as the degree of impact on each component according to the state of the failing component and its vital components. It is based on the supplier-consumer relationship between components. Three possible values may be assigned to an affected component: unaffected (green), partially affected (blue) and blocked (red). At the end, we obtain a picture similar to the one depicted by Figure 1.

It is worth noticing that such algorithms incorporate non-trivial decisions, an example is when a component is simultaneously consumer and producer of several services.

5 Discussion and related work

Identifying vital components for a service is crucial in systems that nest a lot of components with a lot of services. This results many challenging issues, some of them may be addressed by further developing the concepts presented in this paper:

Identifying services that need the same components: this information is not always available at first glance as depending on the designers, it could include more than one design criteria (as architectures are long-living entities that may be changed by many developers). With this respect, proposals could be done to promote reuse by grouping such components.

Identifying degree of coupling between components: in particular identifying strongly coupled components (i.e. components that are vital to each other for almost all the services they provide) during analysis enables users to be knowledgeable that a change or a failure on one component may strongly affect the other. A solution could be to re-architecture the system by merging both components.

Identifying unexpected dependencies and cycles: during analysis, references among components may lead to unexpected dependencies. Those dependencies often should be reconsidered and some could be revoked if they are irrelevant, *e.g.*, a service whose vital components' path is cyclic. Another case is that of components that are strongly mutually consumer/provider to each other.

Change impact analysis: in this paper we limited our analysis to the failure of vital components. In reality the same techniques are usable in an architectural change/evolution context [1]. Each time a component is added, or retracted or (de)composed, etc. Analysis of the impact of such changes [1] is performed to estimate the effect on the system.

Combining different techniques and tools in order to give a more precise understanding: property analysis (model checking, theorem proving), graphical rendering of software architecture (static visualizations, animations), metrics have to be combined and interpreted so that we improve software architecture understanding and diagnostic.

Most of these techniques and tools are based on dependency analysis approaches that are manifold at

the architecture implementation level ([12], [13], [11], [8], [10]) or program slicing based approaches in the context of traditional programming languages [9]. As software architecture is a younger area of investigations, little works focus on dependency analysis of abstract architectures. Aladdin [6] is a tool that helps architects analyzing software architecture by identifying architectural elements that can affect or be affected by another specified architectural element. A chaining technique of dependent component communication ports given in the form of a directed graph is proposed and can be used for studying the system. This proposal can be integrated to some extent with ours, as an alternative for computing the *vital(C,S)* set. Nevertheless the chaining is considered at the component level globally, without taking into account the context of a particular service. As for as traditional slicing [9] techniques, they have been applied to software architectures to support architectural understanding and reuse: static slicing [7] and dynamic slicing [14]. The latter provides an architecture slice of a system that is a component dependency path built using an algorithm based on execution traces. This work focuses on an executing instance of an architecture and therefore may complete the static analysis we target in this paper. In [7], the author proposes a dependency model based on three types of dependencies in an architectural description: component-connector dependencies, connector-component dependencies and additional dependencies that are more specific to the ADL employed. This work can also serve as an alternative for computing the *vital(C,S)* set with as slicing criterion a component's particular service. Nevertheless, the analysis remains limited with regard to vital component identification and the impact of their failure on the system.

All of those works contribute slightly to architectural understanding of software systems by analyzing architectural structure and/or behavior. However they do not support architectural interpretation by, for example, classifying architectural elements as vital elements, non vital, etc. Moreover, in those approaches the ability to decide whether the system can continue working (even in a degraded mode) or not is not supported. We think that software architecture physiology comprehension is a promising area of investigation: providing approaches that contribute to helping software experts (software physiologists) interpreting analysis results and giving diagnostic on a system architecture at different abstraction levels is a new and very challenging issue.

6 References

[1] I. Alloui, Property verification and change impact analysis for model evolution, *1ères journées sur l'Ingénierie Dirigée par les Modèles (IDM'05)*, Paris, 2005, pp. 169-174.

[2] R. Morrison, D. Balasubramaniam, N.C. Kirby,, K. Mickan, F. Oquendo, S. Cimpan, B. Warboys, R. Snowdon, M. Greenwood, Support for Evolving Software Architectures in the ArchWare ADL, *4th Working IEEE/IFIP Int. Conf. on Software Architecture (WICSA 2004)*, Oslo, Norway, June 2004, pp. 69-78.

[3] F. Oquendo, I. Alloui, S. Cîmpan, and H. Verjus, The ArchWare ADL: Definition of The Abstract Syntax and Formal Semantics, ARCHWARE European RTD Project IST-2001-32360, Deliverable D1.1b, 2002.

[4] C. Chaudet, F. Oquendo, π-SPACE: A Formal Architecture Description Language Based on Process Algebra for Evolving Software Systems, *Proc. of 15th IEEE Int. Conference on Automated Software Engineering*, Grenoble - France, September 2000.

[5] R. Allen, R. Douence, D. Garlan, Specifying and Analyzing Dynamic Software Architectures, *In Proc. on Fundamental Approaches to Software Engineering*, Lisbon, Portugal, March, 1998.

[6] J.A. Stafford and A.L. Wolf. Architecture-Level Dependence Analysis for Software Systems. *Int. Journal of Software Engineering and Knowledge Engineering*, 11(4):431–452, August 2001.

[7] J. Zhao. *Using Dependence Analysis to Support Software Architecture Understanding*. New Technologies on Computer Software, pages 135–142, September 1997.

[8] L.A. Tuura, L. Taylor, Ignominy: a tool for software dependency and metric analysis with examples from large HEP packages, *CHEP'01, Computing in High Energy and nuclear Physics*, Beijing,China, September 3 - 7, 2001.

[9] M. Weiser, Program Slicing, *IEEE Transaction on Software Engineering*, Vol. 10, No. 4, pp.352-357, 1984.

[10] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, 2005. Using dependency models to manage complex software architecture. *SIGPLAN* Not. 40, 10 (Oct. 2005), 167-176.

[11] K.P. Pomakis and J.M. Atlee, Reachability Analysis of Feature Interactions: A Progress Report, *Proc. of the Int. Symposium on Software Testing and Analysis*, pp. 216-223, ACM SIGSOFT, January 1996.

[12] J. Ferrante, K.J. Ottenstein and J.D. Warren, The Program Dependence Graph and Its Use in Optimization, *ACM Transaction on Programming Language and System*, Vol. 9, No. 3, pp.319-349, 1987.

[13] J.O. Grady, System Requirements Analysis, *McGraw-Hill*, Inc., 1993.

[14] T. Kim, Y.-T. Song, L. Chung, D. T. Huynh, Dynamic Software Architecture Slicing, *23rd international Computer Software and Applications Conference, COMPSAC*, pp. 61-66, October, 1999.