

A Framework for Agile Development of Component-Based Applications

Guillaume Wagnier, Estéban Duguepéroux, Anne-Françoise Le Meur, Laurence Duchien

Université Lille 1 - LIFL CNRS UMR 8022 - INRIA 59650 Villeneuve d'Ascq, France

Abstract

Agile development processes and component-based software architectures are two software engineering approaches that contribute to enable the rapid building and evolution of applications. Nevertheless, few approaches have proposed a framework to combine agile and component-based development, allowing an application to be tested throughout the entire development cycle. To address this problematic, we have built CALICO, a model-based framework that allows applications to be safely developed in an iterative and incremental manner. The CALICO approach relies on the synchronization of a model view, which specifies the application properties, and a runtime view, which contains the application in its execution context. Tests on the application specifications that require values only known at runtime, are automatically integrated by CALICO into the running application, and the captured needed values are reified at execution time to resume the tests and inform the architect of potential problems. Any modification at the model level that does not introduce new errors is automatically propagated to the running system, allowing the safe evolution of the application. In this paper, we illustrate the CALICO development process with a concrete example and provide information on the current implementation of our framework.

1. Introduction

In many application domains, software systems need to perpetually and rapidly evolve to cope with new user and technology requirements. Being able to modify existing systems or redesign new systems to rapidly take in account new functionalities or preferences has led to the proposition of several software engineering approaches such as the Agile software development methodology [1]. One of the key principles of Agile software development is to build software through an incremental and iterative process. Each iteration adds a new feature and produces a fully working system by going through the whole the software lifecycle, *i.e.*, the analyze, develop and test phases. Another particularity of Agile development is that the testing activity is not just confined to the classical test phase but rather integrated throughout the entire lifecycle, meaning that the software is continuously tested throughout its development, from its specifications to the final running system, in order to augment the overall software system quality.

Another software engineering approach that contributes to facilitating the rapid development of software systems is the use of component-based software architectures. In this context, the overall structure of the application is first described with an architecture description language (ADL) [2]. Such description highlights the needed components and their assembly, which facilitates the understanding and analysis of the application's properties, such as behavioral or quality of service properties. If the specifications are coherent, the application is eventually instantiated, deployed and executed to be tested.

Although Agile software development and component-based software engineering (CBSE) may appear quite different approaches, some works [3, 4] have identified that both approaches could benefit to each other, CBSE bringing for example the capability of building large software and enhancing reusability, and Agile development offering more flexible development processes for shorter time-to-market products. Nevertheless we believe that there is still a bridge between these two approaches, one reason being the lack of component frameworks that allow incremental and iterative development processes, as well as throughout-lifecycle testing.

To address this problematic, we have developed a model-based framework, named CALICO, that enables architects to design and test component-based systems in an iterative and uniformed process [5, 6]. CALICO allows software architects to specify their architectures as models, and to analyze them with respect to application and platform constraints. Our approach enables the testing of the system throughout the system lifecycle. More concretely,

CALICO analyses architecture models and creates contracts by composing contractual application properties, *e.g.*, behavioral, dataflow, QoS properties. This composition allows compatible and incompatible interaction to be identified, as well as partially compatible interactions, which require runtime checking [7]. When runtime checking is needed, CALICO automatically instruments the application to reify runtime information to complete the resolution of the partially compatible interaction contract and thus detects if the given interaction may lead to an error. By using this framework in iterative software design processes, architects get design feedback, *i.e.*, information on identified interaction errors, and can then modify the models accordingly. Each modification performed on the model is propagated to the running system since CALICO ensures the synchronization between the model and the runtime system, both of which thus coexist during the whole application development. Furthermore, the solution offered by CALICO is generic regarding underlying platforms, allowing component platforms to benefit from all the analyses integrated into CALICO.

The rest of this paper is organized as follows. Section 2 gives an overview of the CALICO iterative and incremental development process. Section 3 illustrates with a concrete scenario the CALICO approach. Finally, Section 4 provides some information about the current status of our framework implementation.

2. CALICO Overview

CALICO is composed of two levels: a model level and a platform level as shown in Figure 1. The model level is independent of any component-based or service-oriented platform. It contains the CALICO Architecture Description metamodels that enable an architect to describe the structure and the properties, *i.e.*, structural, behavioral, dataflow and QoS properties, of an application. It is also possible to specify some contextual adaptation rules, independently of any platform, in order to allow the debugging of autonomic systems. The platform level holds the running system on a target platform.

The iterative and incremental development process of CALICO, illustrated in Figure 1, is as follows :

(1) Design : The architect specifies the design of the desired application using the CALICO metamodels. The system structure metamodel enables architects to describe the structure of their architecture, independently of any component platform. CALICO provides also four contract metamodels to allow architects to specify structural, behavioral, dataflow and QoS properties for each component.

(2) Static analysis : The interaction analysis tool checks the coherence of the system architecture. For each partially compatible interaction, a test to be performed at execution time is automatically inserted into the CALICO debug metamodel. For each incompatible interaction, the architect is notified of the problem and he/she may thus provide some modification of the application design. As long as some incompatible interactions remain, the next steps of the development process can not be reached. Once all of the problems are fixed, the architect specifies the runtime platform on which the application is to be executed and CALICO verifies that the specifications do not go against the platform constraints in order to make sure that the application can be indeed deployed on that specific platform.

(3) Code generation : If a component or service does not already exist, then the generation tool generates code skeleton such that only business code needs to be provided by the developers.

(4) Instrumentalisation : This step makes the link between the static analysis and the dynamic checks of the application at runtime. The instrumentation tool takes the debug model as input and automatically instruments the application code to enable the capture of the needed runtime information to complete the resolution of the partially compatible interaction. This instrumentalisation relies on an aspect-oriented approach and is independent of the underlying platform.

(5) Instantiation : The loader instantiates the application on the target platform as described by the architect's structural model. Concretely, the running system is created incrementally by calling the appropriate sequence of system construction operations, such as creating/removing components and connectors.

(6) Reification : As the testers run the application in different execution contexts, the instrumented application automatically reifies any context changes and monitored information.

(7) Dynamic debugging : During the debugging phase, the debug tool analyzes the information reified by the running system and triggers when needed the tests contained in the debugging model. The architect is notified each time an error is detected, allowing him/her to correct the application design. Other debugging action rather than the notification action maybe chosen, such as logging the information into a file, or executing a reconfiguration script that

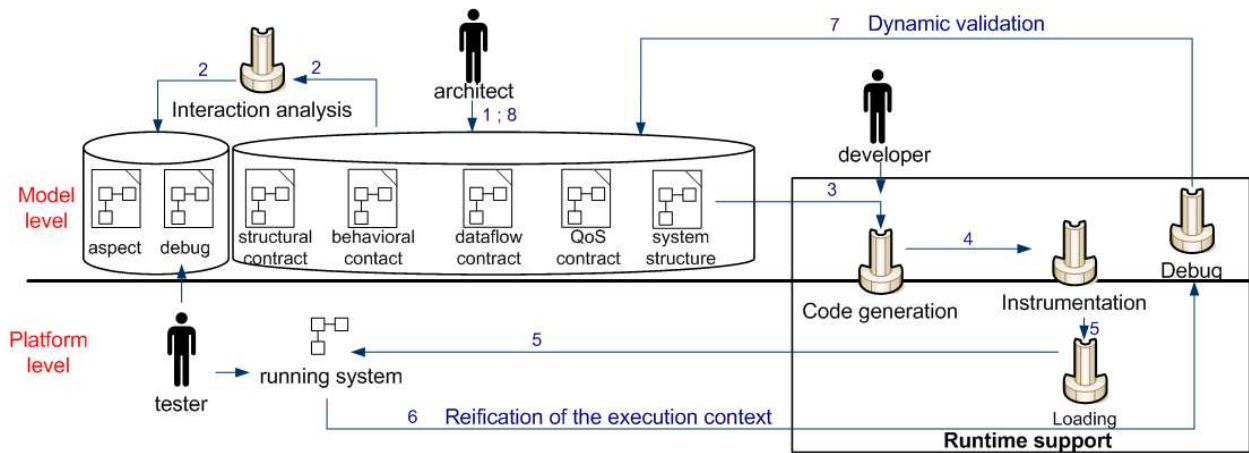


Figure 1: Overview of CALICO iterative development process

will automatically modify the design and trigger the step 2 of the process. This latter case may be useful to tune/test adaptation policies for autonomic system.

(8) Evolution of the design : The architect can modify the design with respect to the debugging information if problems have occurred. They can also adapt at anytime the design of the application to address new user or application requirements. After any modification, the development cycle iterates again starting at step 2.

3. Illustrative Example

To illustrate the agile development process offered by CALICO, we use an example of architecture in the context of the French Personal Health Record system (PHR) [8]. PHR is the French personal health record system that is intended to provide health-care professionals with the information needed for their patients care. Figure 2 represents a possible architecture of the PHR system. All medical information, (such as biological analyses, X-rays, medications, etc.), will be stored in distributed databases and will be made accessible through an on-line interface Client.

In order to build a robust PHR application, architects need to be able to express several application properties. A first requirement of this system architecture is related to authentication issues since not everybody should have access to anybody's health records. The architecture of this system must thus provide some authentication mechanism. The Authentication architecture element logs a health-care professional in and returns a session ticket through the functionality `getTicket` that is offered by `SessionServer`. For security reason, the functionality `getTicket` can be used only by the element Authentication to avoid that an unauthenticated user get a session ticket. Finally the session ticket must be validated by the `SessionServer` before retrieving any medical data from the database.

Another requirement is the high reliability of the system. Such system has to be able to handle very heterogeneous medical information, going from light-weight text records to gigabytes of echographies. Furthermore, the devices used to display this information are also heterogeneous. They range from desktop computers with high-quality large-screen monitors and gigabyte network connexions to simple PDAs with small screens and low-bandwidth GPRS network connexion. Handling such data in a reliable way is critical because the system must be able to determine if a given data can be displayed appropriately with no loss of information, as well as in which time-frame, depending on the available resources and amount of information to display. For example, a dataflow constraint may express that medical data received on a terminal of type PDA Nokia N800 should be less than 10 megaoctets. Another constraint can also specify that only text or jpg documents can be read on that terminal.

Overall the constraints may evolve in time or just not reflect exactly one given execution context. There is thus a need to iterate the whole process to check if the declared application constraints can be all checked, statically or dynamically.

(1) The architect specifies the architecture and the properties of the PHR application. For example, the first requirement mentioned above can be specified using the structural contract metamodel.

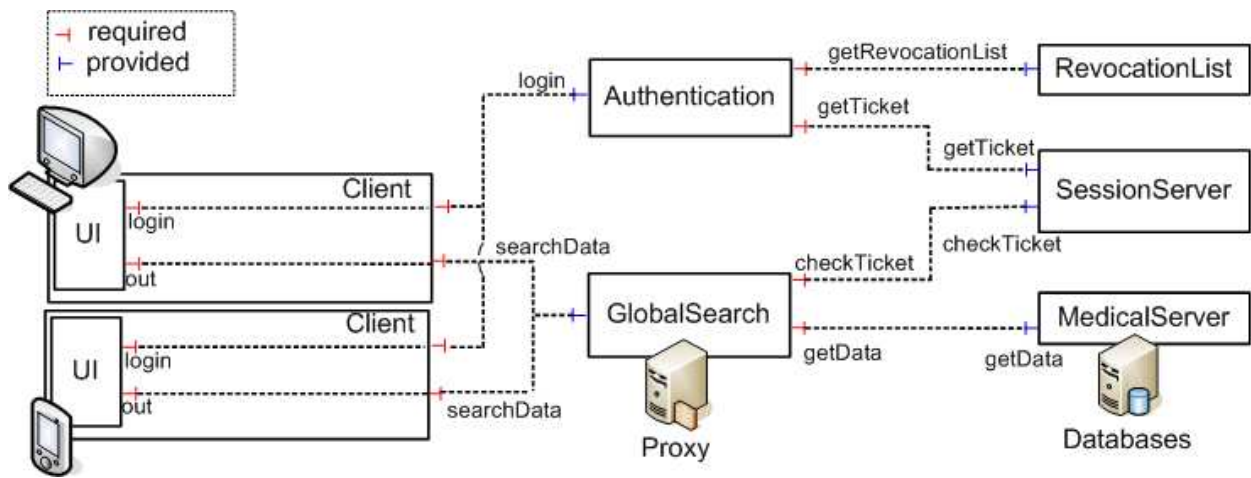


Figure 2: Structure of the PHR application

(2) The overall coherence of the constraints is statically verified. A partially compatible interaction is detected between the MedicalServer and the PDS since data sent by the server could be greater than 10 megaoctets or in a different format than txt or jpg. Accordingly CALICO adds some rules to validate in the debug model. These rules specify that the size and the data type must be captured at runtime.

(3) Code skeletons are generated and developers can provide the business code of each components.

(4) Following the information contained in the debug model, the application is automatically modified to capture the size and the type of the medical data that enters the PDA.

(5) The application is deployed on the target application.

(6) At this step different execution contexts are tested. One may consists in the use of the PHR application by a druggist, who typically uses the PHR only to consult text documents. Another test scenario considers a radiologist. During the test scenario execution, monitored informations are reified.

(7) The debug tool resumes the interaction compatibility checks that were partially compatible. In the case of the druggist, no error is detected, whereas for the radiologist, the analyse indicates that the data are too large for the PDA.

(8) The architect can accordingly modify the application design by inserting a new component DataConverter between the PDA and the GlobalSearch component in order to reduce the size of a too large radiography.

The whole process is then iterated again. If no error is detected statically, the new component DataConverter is automatically integrated into the already deployed application, and new test scenarios may be executed.

4. Conclusion and Current Implementation Status

CALICO is a model-based framework that enables the design and debug of systems in an iterative and incremental way, bridging a little more the gap between component-based software engineering and agile development approaches. Our framework is generic and highly extensible. All metamodels for specifying the structure, the application properties and the adaptation rules are independent of any underlying platform. This enables architects to perform various architecture analyses on their applications even if the underlying component or service framework does not provide any verification tools.

The current implementation of CALICO is developed in Java. All CALICO metamodels are implemented with the Eclipse Modeling Framework (EMF). A graphical editor, implemented with the Graphical Modeling Framework (GMF), enables the architect to edit the model during the whole development cycle.

We have integrated several existing tools to verify the coherence of the component interactions in term of structural, behavioral, dataflow and quality of service properties. Structural constraints are expressed in OCL [9], using the EMF-OCL library. Behavioral specifications are based on existing process algebra, such as CSP [10], FSP [11], SFSP. The current implementation uses the Fractal behavioral protocol checker [12] to verify that a given component

composition does not introduce a deadlock. We have developed a dataflow analysis based on the algorithm of constant propagation in partial program validation [13]. The QoS metamodel has been inspired by the QML [14] and WSLA [15] approaches. The associated analysis is based on prediction of quality property in a workflow of WEB services [16]. Furthermore, application instrumentation has been implemented with Spoon [17]. The sensor framework Wildcat [18] has been integrated in CALICO. Our current implementation supports four component platforms (Fractal [19], OpenCCM [20], OpenCOM [21] and FraSCAti [22]) and one service-oriented platform (Web services [23]). CALICO has been carefully designed to allow new extensions in terms of support for new platforms, new QoS sensors and new kinds of debugging actions.

We have performed benchmarks on our implementation and showed that CALICO is usable to design reliable large systems up-to 10000 components, which is the maximum load of most runtime platforms.

CALICO is still being developed, to support more extensions. The current implementation is freely available at <http://calico.gforge.inria.fr>.

References

- [1] Alliance, A.: Manifesto for agile software development (2001) <http://agilemanifesto.org>.
- [2] Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. *IEEE Trans. on Software Engineering* **26**(1) (January 2000)
- [3] Stojanovic, Z., Dahanayake, A., Sol, H.G.: Component-oriented agile software development. In Marchesi, M., Succi, G., eds.: *XP*. Volume 2675 of *Lecture Notes in Computer Science.*, Springer (2003) 315–318
- [4] Radinger, W., Goeschka, K.M.: Agile software development for component based software engineering. In: *Proceedings of 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, ACM (October 2003) 300–301
- [5] Waignier, G., Sriplakich, P., Le Meur, A.F., Duchien, L.: A Model-Based Framework for Statically and Dynamically Checking Component Interactions. In: *Proceedings of the ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MODELS'08)*. Volume 5301 of *Lecture Notes in Computer Science.*, Toulouse, France, Springer-Verlag (October 2008) 371–385
- [6] Waignier, G., Le Meur, A.F., Duchien, L.: A model-based framework to design and debug safe component-based autonomic systems. In Mirandola, R., Gortona, I., Hofmeister, C., eds.: *Proceedings of the 5th International Conference on the Quality of Software-Architectures (QoSA 2009)*. Volume 5581 of *Lecture Notes in Computer Science.*, Pennsylvania, USA, Springer-Verlag (June 2009) 1–17
- [7] Waignier, G., Le Meur, A.F., Duchien, L.: Architectural Specification and Static Analyses of Contractual Application Properties. In Becker, S., Plasil, F., Reussner, R., eds.: *Proceedings of the 4th International Conference on the Quality of Software-Architectures (QoSA'08)*. Volume 5281 of *Lecture Notes in Computer Science.*, Karlsruhe (TH), Germany, Springer-Verlag (October 2008) 152–170
- [8] Nunziati, S.: Personal health record www.d-m-p.org/docs/EnglishVersionDMP.pdf.
- [9] Object Management Group: *Object Constraint Language (OCL)*, Needham, MA, USA. 2.0 edn. (May 2006)
- [10] Hoare, C.: *Communicating Sequential Processes*. Prentice Hall (June 2004)
- [11] Magee, J.: Behavioral analysis of software architecture using ltsa. In: *Proceedings of the 21st international conference on Software engineering*, IEEE Computer Society (1999) 634–637
- [12] Barros, T., Ameur-Boulifa, R., Cansado, A., Henrio, L., Madelaine, E.: Behavioural models for distributed fractal components. *Annals of Telecommunications* **64**(1/2) (February 2009) 25–43
- [13] Kildall, G.: A unified approach to global program optimization. In: *1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. (1973)
- [14] Frolund, S., Koisten, J.: *QML: A Language for Quality of Service Specification*. (1998)
- [15] IBM: *WSLA Language Specification, V1.0*. (2003)
- [16] Xiangpeng, Z., Chao, C., Hongli, Y., Zongyan, Q.: A qos view of web service choreography. *IEEE International Conference on e-Business Engineering* (2007) 607–611
- [17] Pawlak, R.: Spoon: Compile-time annotation processing for middleware. *IEEE Distributed Systems Online* **7**(11) (nov 2006)
- [18] David, P.C., Ledoux, T.: Wildcat: a generic framework for context-aware applications. In: *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, ACM (2005) 1–7
- [19] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: An open component model and its support in Java. In: *Proceedings of the 7th International Symposium Component-Based Software Engineering*. Volume 3054 of *Lecture Notes in Computer Science.*, Edinburgh, Scotland, Springer-Verlag (May 2004) 7–22
- [20] Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, J.: *Opencom v2: A component model for building systems software*. In: *Iasted Software Engineering and Applications*. (November 2004)
- [21] Briclet, F., Contreras, C., Merle, P.: OpenCCM : une infrastructure à composants pour le déploiement d'applications à base de composants CORBA. In *IMAG/LSR, ed.: Proceedings of the 2004 Déploiement et (Re) Configuration de Logiciels (DECOR'04)*. (2004) 101–112 <http://openccm.ow2.org>.
- [22] Seinturier, L., Merle, P., Fournier, D., Dolet, N., Schiavoni, V., Stefani, J.B.: Reconfigurable sca applications with the frascati platform. In: *6th IEEE International Conference on Service Computing (SCC'09)*. (September 2009)
- [23] Erl, T.: *Service-oriented Architecture: Concepts, Technology, and Design*. Prentice Hall (2005)