Providing Support for Safe Software Architecture Transformations

Olivier Barais*, Julia Lawall†, Anne-Françoise Le Meur* and Laurence Duchien*

*Jacquard project, INRIA/LIFL
Université des Sciences et Technologies de Lille
59655 Villeneuve d'Ascq Cedex, France
{barais, lemeur, duchien}@lifl.fr

†DIKU, University of Copenhagen
2100 Copenhagen Ø, Denmark
julia@diku.dk

1. Introduction

Software architecture is a key concept in the design of a complex system. An architecture models the structure and behavior of the system, including the software elements and the relationships between them. While architectures were originally specified informally, recent years have seen the creation of a number of Architecture Description Languages (ADLs) [4]. ADLs are designed around the dimensions of composition and interaction, allowing the architect to introduce new concerns by constructing and combining increasingly complex elements.

Simple composition and interaction are sufficient as long as new elements match the interface provided by the existing architecture. Some concerns, however, such as security, crosscut the software architecture and cannot be expressed in a modular way. To integrate such concerns, the architect must invasively modify the architecture elements and the connections between them, at all points affected by the concern. These modifications are low-level, tedious, and error-prone, making the integration of new concerns difficult.

To address the complexity of integrating a new concern into a software architecture, we have developed the TranSAT framework. Starting with a core architecture containing some business concerns, the architect uses TranSAT to incrementally add all needed technical and business concerns. Inspired by Aspect Oriented Programming [3], TranSAT isolates the description of each concern in a separate architecture construct, the *pattern*, that is automatically integrated with the core software architecture by a weaver. This pattern consists of a new architecture fragment to be integrated and a description of where it can be applied, as well as a sequence of transformation rules describing modifications to the existing architecture.

In a previous paper, we have presented the global design of TranSAT [2]. This paper identifies some of the problems that an architect may encounter when manually integrating a new concern, and then gives an overview of the features of the TranSAT framework that contribute to ensuring the consistency of software architectures created through successive automatic concern integrations.

2. Some Manual Concern Integration Issues

Integrating a new concern into an existing software architecture involves modifying the component structure, behavior, and interfaces. This task is highly error prone, as many modifications may be required. We examine this issue in the context of the SafArchie component model [1], which is the target of TranSAT and contains many standard features. In particular, a SafArchie architecture consists of a collection of possibly nested components, combined with a collection of bindings that connect components at ports and represent imported and exported operations. Additionally, each component is associated with an input/output automaton that describes its behavior in terms of these operations. In such a component model, integrating a new concern can have both a local impact on the modified elements and a global impact on the consistency of the architecture as a whole.

Local impact SafArchie places a number of requirements on the various architectural elements. For example, a port must contain at least one operation, must be bound to exactly one other port, in some other component, and this other port must provide compatible operations. When integrating a new concern, an architect typically adds bindings between the components of the new concern and the components of the existing architecture. These new bindings must respect the requirements on ports.

Modifications to the behavior automaton associated with each component are particularly error prone, because the automaton must be kept coherent with the other elements of the component and because of the complexity of the automaton structure. All of the operations associated with the ports of a component must appear somewhere in the component's behavior automaton. As SafArchie separates the structural and behavioral descriptions, it is easy to overlook one when adding or removing operations from the other. An automaton must also describe a meaningful behavior. Checking the behavior requires tracing through all of the paths of the automaton, which can be difficult to do manually. Finally, removing an operation requires removing it from wherever it appears in the automaton. This may involve reorganizing the automaton to eliminate paths that are

no longer meaningful, which can be a complex transformation.

Global impact So that the application can run without deadlock, it must be possible to synchronize the behavior of each component with that of its neighbors. Any change in the behavior of a single component can affect this synchronization, which can in turn affect the neighbors' synchronization with the rest of the architecture. The interdependencies between behaviors can make the source of any error difficult to determine.

3. Safe Software Architecture Integration

In TranSAT, a concern is represented as a *plan*, a *join point mask*, and a set of *transformation rules*. The plan describes the structure and behavior of the new concern. The join point mask defines the structural and behavioral requirements that the basis plan must satisfy so that the new concern can be integrated. The transformation rules specify the means of integrating the new plan with the basis plan. In these terms, the issues identified in Section 2 suggest what can go wrong in the transformations that connect the new architecture fragment specified by the plan to the existing architecture fragment specified by the join point mask. TranSAT prevents these errors through a combination of constraints on the transformation language and static and dyanmic verifications.

Restrictions on the transformation language Many of the local errors identified in Section 2 occur when a new element affects multiple elements in the existing architecture. The TranSAT transformation language prevents some of these errors by providing abstractions that update all affected elements at once, in a consistent manner. For example, replacing an operation in a port requires updating both the port structure and the automaton of the associated component. Consequently, the transformation language combines both in a single `replace` transformation operation.

Static verifications Other local properties are affected by multiple transformation steps, and thus ensuring these properties requires static analysis of the relationships between transformation operations. As an example, we consider the requirement that a port contain at least one operation. This property is affected by the totality of the transformations that add or remove an operation from the given port. TranSAT statically simulates the execution of the transformation rules on the various elements identified by the plan and the join point mask. At the end of the analysis, every port must be proved to have at least one operation.

Dynamic verifications Because the join point mask does not describe the entire basis architecture, the information in a pattern is not sufficient to ensure that the behaviors of the various components of the resulting architecture can be synchronized. Indeed, adding new components and behaviors to a fragment of an architecture can change the synchronization at the interface of the fragment, and thus have an effect on the synchronization of the rest of the architecture. The use of TranSAT localizes the modifications to a specified fragment of the existing architecture. Resynchronization at transformation time starts from the affected fragment and works outward until reaching a composite for which the interface is structurally unchanged and the new automaton is bisimilar to the one computed before the transformation.

4. Assessment and Future Work

When an architecture is updated manually, it is only possible to verify its consistency once the transformation is complete. Thus, if an error is detected, the architect has to manually retrace his work to separate the correct modifications from the erroneous ones. The language restrictions and static verifications of TranSAT make it possible to ensure many safety properties *a priori*, before the existing architecture is modified. Although the remaining properties are verified dynamically during the transformation process, TranSAT records enough information to allow it to roll back to the untransformed version if errors are detected. These features allow the architect to easily experiment with new variants and make it possible to use a pattern provided by a third party developer with confidence that the concern will be integrated correctly.

Currently, TranSAT targets SafArchie, and thus the verifications associated with the transformation process are mainly derived from the constraints imposed by the SafArchie model. In future work, we would like to decouple TranSAT from SafArchie, to target other ADLs, guided by their safety requirements.

References

[1] O. Barais and L. Duchien. SafArchie studio: An ArgoUML extension to build safe architectures. In Pierre Dissaux, Mamoun Filali Amine, and Pierre Michel, editors, *Architecture Description Languages*, pages 85–100. Springer, 2005.

[2] O. Barais, L. Duchien, and A.-F. Le Meur. A framework to specify incremental software architecture transformations. In *31st EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 62–69. IEEE Computer Society, September 2005.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In Mehmet Akcsit and Satoshi Matsuoka, editors, *Proceedings ECOOP*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[4] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.