Interaction-sensitive Synthesis of Architectural Tactics in Connector Designs

Thorsten Keuler, Christian Webel

Fraunhofer Institute for Experimental Software Engineering, Fraunhofer-Platz 1

67663 Kaiserslautern, Germany

{thorsten.keuler, christian.webel}@iese.fraunhofer.de

Abstract

During architectural design, the architect has to come up with architectural structures and tactics that aim at the fulfillment of quality attribute requirements. Architectural structures are built from component and connector types that define how components are supposed to interact. Since connector types typically impact big portions of a system, tactics usually crosscut connector designs and demand for modularized treatment during architecture design. The synthesis of tactics within connector designs, however, turns out to be a major challenge. This is due to the fact that they are likely to affect each other in the final system where they need to be mutually integrated. This mutual affection is called tactic interaction. In this paper, we describe an approach towards detecting such tactic interactions during connector design. Our approach is integrated in a commercial architecture design tool supporting interaction-sensitive synthesis of architectural tactics during connector design activities.

1. Introduction

During architectural design, the architect has to come up with a set of architectural tactics that aim at the fulfillment of non-functional requirements. The selection and application of an appropriate set of such tactics is crucial, since the architecture of a system influences all subsequent development activities, even beyond deployment. In that context, software architecture can be described by a number of interconnected and structured components, and architectural connectors that describe how these components interact [1]. Connectors can be formally described by a set of roles representing interacting entities and a glue specification that describes the interplay of the different collaborating roles [2]. However, it is the tactics that usually crosscut the interaction specifications of connectors and thus demand for modularized treatment during architecture design [3]. The synthesis of tactics within connector designs, however, turns out to be a major challenge. This is due to the fact that even though tactics are designed separately, they are likely to influence each other in the final system where they need to be mutually integrated. These unwanted compositional effects are also known as "interactions" [4]. Consequently, in the context of architectural connector design, tactic interactions need to be detected and resolved in order to ensure that composed connectors exhibit all properties desired. The work presented in this paper shows an effective and efficient way of how to detect tactic interactions in architectural connectors.

2. Approach Overview

From an overall perspective, our approach for connector design is embedded in a four-step modeling process as depicted in **Figure 1**:

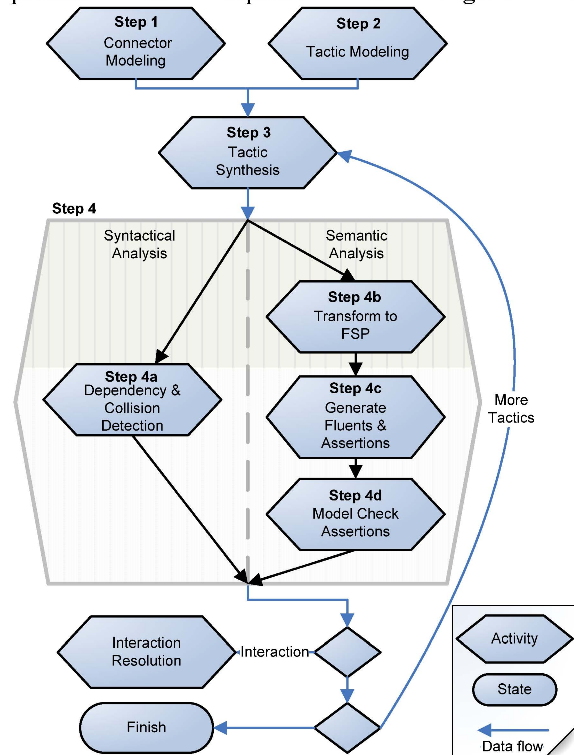

Figure 1 Overview of our Approach

<u>Step 1:</u> In the first step, we design the initial connector model. A connector model comprises sequence diagrams that model interaction behavior between communicating entities (roles).

978-1-4244-4985-9/09/$25.00 ©2009 IEEE

Step 2 is concerned with the modeling of tactics. As the connector models, we define tactics in terms of sequence diagrams as well.

Step 3: The synthesis itself follows the ideas of aspect-orientation [5] and is tool supported as it is described in [7]. A central notion in aspect-orientation is the so-called join-point model (JPM). Basically, a JPM defines three things:

(1) **Join-point**: A join-point is a point in a model where behavior can be altered, augmented or removed. In the context of connector design, a join-point would be a place in the interaction specification that can be altered, augmented or removed by a tactic.

(2) **Pointcut**: A pointcut is a predicate over a set of join-points. In other words, pointcuts select sub sets of model elements and thus determine where tactics should be applied.

(3) **Advice Directive**: In general, there exist three different advice directives denoting the way additional behavior is introduced in a connector model: *before*, *after*, and *around* [4].

Step 4: In this step, we analyze the synthesized models with respect to interactions. In case we detect an interaction of any kind, we proceed to the interaction resolution. Resolution strategies, however, are not in the focus of this paper. The details of step 4 are explained in the next section.

4. Interaction Detection

In general, interactions can either be on the syntactical *(collisions)* or on the semantic *(conflicts)* level. *Collisions* might arise in case one tactic is applicable to a model element introduced by another tactic or in case two tactics operate on the same elements in the base communication model. Collisions can be detected syntactically whereas conflicts need to be detected on the semantic level.

Step 4a: For syntactical analyses, we analyze the join-point collections of two tactics with respect to each other and the connector model. Due to already existing solutions syntactical analyses are not in the focus of this paper though.

Step 4b: For detecting conflicts, we apply fluent linear temporal logic [6] in order to capture the semantic properties that are to be preserved after synthesizing tactics in a connector design. In order to check semantic properties of composed models, we specify properties in the realm of event-based systems. In other words, regarding actions, events, or message occurrences, we define assertions about events, or sequences of events, that should never arise or always be true. By assertions we refer to state-based temporal logic properties specified over an event-based operational model that have to be satisfied after tactics have been synthesized. A fluent is a time-varying property that is true at some time instant t if it has been initiated by an action or message at a time instant t_p prior to t and not terminated by another action or message in the meantime. More formally, a fluent F is a proposition defined by a set of initiating actions *Init* and terminating actions *Term,* and an optional attribute *Initially* that states whether the fluent F is true or false at time zero:

$$F =_{DF} \langle \, Init, Term \, \rangle \text{ with } Init,\, Term \subset Action$$

By properties we refer to properties of the tactics as well as those of the connector model itself. For instance, we can define assertions that stipulate having *message* m_1 always followed by *message* m_2. Eventually, we transform the synthesized model from step 3 into FSP (Finite Sequential Processes).

Step 4c: In this step, we generate fluents and assertions based on the tactics that were modeled in step 3. For deriving assertions we need to know in what way the tactics are supposed to be synthesized within the connector model. This is due to the fact that the way tactics are synthesized (before, after, around) implies contextual properties of that tactic with respect to the connector model. The derived assertions capture these contextual properties and make them checkable during further modifications by other tactics. In the following, we provide some basic heuristics for deriving such kinds of FLTL assertions and fluents from tactics specified in the form of sequence diagrams automatically.

Let T be a tactic and P be the corresponding pointcut. When defining an assertion that describes the desired behavior of a connector model augmented by tactic T, we have to distinguish between two cases: synchronous interaction and asynchronous interaction at the join-point. In the following, let *join-point* denote the message where the connector model and the tactic meet. For synchronous communication, let *join-point_rsp* denote the return message caused by a synchronous message *join-point*.

Advice-Directive: **Before**

A feasible assertion for the before-advice is given by

$$\Box\,(join\text{-}point \rightarrow \text{BeforeAdvice}), \text{ with fluent BeforeAdvice} = \langle\, validatorMsg, join\text{-}point_rsp \,\rangle \quad \text{(A1)}$$

where $validatorMsg \in Msg_T$ is a message that validates the fluent and satisfies the assertion at the join-point.

Example: A simple *Billing* tactic defines a new role *bill-Role* that is in charge of handling the Billing coordination within a connector. The *bill-Role* is triggered via the *bill* message. In this example, we want to assure that any message of the type <<call>> in the connector model can only be submitted if the Billing tactic has been applied before. Therefore, the

advice directive for introducing Billing into the connector automaton is **before() : call → Billing**. Consequently, a generic assertion for the Billing advice may be defined according to (A1) as □(*call* → Billing), with Billing being one of the following two fluents

fluent Billing_Sync = ⟨*bill*, *joinpoint_rsp* ⟩

After applying the pointcut to the connector automaton, we can replace the *joinpoint_rsp* message with corresponding messages of the connector model (here: *register*, see **Figure 2**), leading to the following assertions and fluents for interactions of the type <<call>>:

□ (*register* → BillingReg), with the fluent Fluent BillingReg = ⟨ *bill, register_rsp*⟩

Since *register* describes a synchronous interaction, the fluent is invalidated by the synchronous response message *register_rsp* (A1).

In general, when interacting asynchronously, there is no response message that can be used to invalidate the fluent. For that reason, all messages $m = (p, s, src, trgt)$ that may occur directly after the *join-point* are used. Let $Join\text{-}pointNext = \{m \in Msg_C{}^{1} \mid p = join\text{-}point\}$. A feasible assertion for asynchronous communication can then be defined as follows:

□ (*join-point* → BeforeAdvice), with fluent BeforeAdvice = ⟨ *validatorMsg*, *Join-pointNext* ⟩	(A2)

The just defined assertions are, in a sense, not very strong, but sometimes, strict coupling is mandatory. For instance, if for security reasons, an authorization is required *immediately* before a login, without any further events or messages in between. In that case, the set *JoinpointNext* has to be extended by all messages $\in Msg_T$ that belong to the tactic and may occur between *validatorMsg* and *join-point.* Therefore, let o be a non-empty sequence of messages of length n denoting the allowed order of message exchanges over T and let m_i denote the i-th message of that sequence. Then, $\forall\, i < n$ is $m_i = p_{i+1}$. Let $Ind_o : Msg \to \mathbb{N}$ be an index function over o where $Ind_o(m)$ denotes the position of m in the sequence o and $j = Ind_o(validatorMsg)$ and $k = Ind_o(join\text{-}point)$, respectively. For strong assertions, the set of messages Inv that invalidate the fluent is defined as $Inv = JoinpointNext \cup Msg^*_T \cup Msg^*$, where $Msg^*_T = \{m \in Msg_T \mid Ind_o(m) < Ind_o(validatorMsg)\}$ and $Msg^* = \{m \in Msg_{A1} \cup \ldots \cup Msg_{Tn} \mid m \neq join\text{-}point \wedge m \neq validatorMsg \wedge m \notin Msg^*_T\}$.

The latter describes the set of messages defined by other tactics to be injected, and thus might be inserted in between the validating message of T and the join-point. So the resulting fluent for a before-advice is given by

fluent BeforeAdvice = ⟨ *validatorMsg*, *Inv* ⟩ (A3)

In case of asynchronous communication, a connector model with two succeeding messages of the same type would satisfy the above assertions, although it is obviously violated. This is due to the fact that when the second message occurs, the fluent is still valid, since no response message *"_rsp"* will invalidate it.

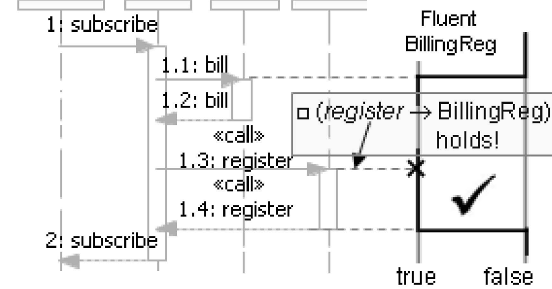

Figure 2 Assertion holds in the synthesized model

However, this can only happen if tactics depend on each other or if they are composed incorrectly. The former can be detected statically without synthesizing; the latter may occur if the composition algorithm is incorrect. A more precise assertion avoiding the aforementioned drawbacks can be defined by using the next operator or a global clock such that

□ (*join-point* → BeforeAdvice ∧ ○¬*join-point*), or □ (*join-point* → BeforeAdvice ∧ $\lozenge_{\leq 1}$ ¬*join-point*)	(A4)

Obviously, it is impossible to successfully enhance a connector model by two different before-advices whose assertions are strong, since the resulting model cannot satisfy both properties at once.

Advice-Directive: **After**

Assertions for an after-advice for synchronous interactions can be defined as follows:

□ (*join-point_rsp* → (¬(m_1 ∨ … ∨ m_n) **W** AfterAdvice)), with AfterAdvice = ⟨{$valid_1,\ldots,valid_n$},{$invalid_1,\ldots,invalid_m$}⟩	(A5)

The above assertions state that after every occurrence of the *join-point* message, the advice has to be executed before any of the messages $m_1,\ldots,m_n$ occurs. The messages m_i are to be chosen according to the context. For instance, a strong assertion can be derived from the above by choosing all $m_i \in Msg \setminus \{jointPoint_rsp\}$. The messages $valid_i$ and $invalid_j$ are appropriate messages taken from the tactic, i.e., $\forall\, i,j\ .\ valid_i, invalid_j \in Msg_T$.

In the asynchronous case, *join-point_rsp* has to be replaced by *join-point*, since no explicit return message exists. As in the before-advice, a connector model with two succeeding messages corresponding to join-points, but only one after-advice, satisfies the above assertion, although the model does not show the desired

[1] Msg_c are all messages defined within the connector

behavior. A more sophisticated assertion can be defined in the following two ways:

$$\square(\textit{join-point} \rightarrow (\neg(m_1 \vee \ldots \vee m_n)\mathbf{W}\text{AfterAdvice})) \wedge \circ\neg\textit{join-point}\text{, or}$$
$$\square(\textit{join-point} \rightarrow (\neg(m_1 \vee \ldots \vee m_n)\mathbf{W}\text{AfterAdvice})) \wedge \Diamond_{\leq 1}\neg\textit{join-point} \quad \text{(A6)}$$

Advice-Directive: **Around**

An around-advice replaces any join-point with the interactions described by the tactic, either by keeping the join-point and adding some "interactions" before and after, or by completely substituting the join-point. For the first case, the around directive can be seen as a sequential composition of appropriate before- and after-directives. A feasible assertion is given by:

$$\square(\textit{BeforeAssert} \wedge \textit{AfterAssert}) \quad \text{(A7)}$$

where *BeforeAssert* and *AfterAssert* are corresponding assertions for a before- and an after-directives as described in the previous subsections. Further, the join-point may also be renamed, that is, the join-point does not necessarily have to be in Msg_C, but can also be in Msg_T. In the second case, the presented heuristic cannot be used to (automatically) derive fluents and assertions based on the interaction advice. This is due to the fact that in the synthesized model, there is no particular trigger or message coupled with the tactic and therefore the assertions cannot be specified without knowing the embedding context. In that case, the assertion has to be defined manually. Eventually, for every join-point that is modified by a tactic, we generate one individual assertion that reflects the properties of the tactic in that particular join-point context.

Step 4d: In the last step, we run the LTS Analyzer [8] in order to check if the defined assertions hold for the augmented connector model. For this purpose, the FSP specification (Step 4a) and the assertions (Step 4b) serve as input to the tool. The LTSA uses the FSP description to generate a LTS behavior model that is equivalent to our connector model. Finally, we run the LTS analyzer and check if the defined assertions hold for the augmented connector model.

5. Related Work

The seminal paper that our work is built on is a paper by Allen and Garlan [2]. Our work takes the notion of a connector as defined by those authors and extends their work mainly in terms of construction support throughout the design process. The motivation for our work stems from the challenges that arise in the context of the iterative design of connectors. Our main contribution, namely the heuristics for automatically deriving assertions and fluents for model checking the integrity of synthesized tactics in connector models, conceptually and technically extends existing work in the area of aspect interaction detection. Concerning interaction detection during the composition of systems from aspects, a number of approaches already exist. However, most of the work concentrates on the syntactical level, i.e., on potential conflicts between aspects caused by the syntax. Our solution combines existing approaches in such a way that effort-intensive and error-prone manual tasks during connector design can be automated.

6. Summary & Outlook

The main contribution of this paper is the heuristics for generating assertions and fluents from tactic designs in an automated fashion. The assertions and fluents are used for model-checking a synthesized connector model that has been augmented by a set of potentially interfering tactics. The synthesis of tactics as well as the transition to FSP are fully automated and integrated in a commercial architecture design tool. Concerning future work we will apply the approach to models of higher complexity to gain a better understanding of potential influence factors that our approach needs to be aligned with. Besides, we also need to take into consideration that the adaptation of assertions or fluents is a continuous process during the incremental design of a connector. Assertions or fluents might need to be loosened in one case or the other, depending on the local composition rules of tactics. In addition, we will have to investigate interaction resolution strategies in more detail. Especially the resolution of semantic conflicts is likely to require sophisticated methods and techniques in order to support an efficient development process overall.

7. References

[1] M. Shaw, D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, Upper Saddle River NJ, USA, 1996

[2] R. Allen, D. Garlan, *A formal basis for architectural connection*. ACM Trans. Softw. Eng. Methodol. 6, 3 (Jul. 1997), pp. 213-249.

[3] R. Filman, S. Barrett, D. Lee, T. Linden, *Inserting Ilities by Controlling Communications*, In Aspect-oriented Software Development, pp. 283-295, 2005

[4] Douence, R., Fradet, P., Südholt, M. 2002. *A Framework for the Detection and Resolution of Aspect Interactions*. GPCE, 2002.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J. Irwin, *Aspect-Oriented Programming*, ECOOP, 1997.

[6] Giannakopoulou, D., Magee, J. *Fluent model checking for event-based systems*. ESEC, Helsinki, Finland, 2003.

[7] Keuler, T., Muthig, D., Uchida, T. *Efficient Quality Impact Analyses for Iterative Architecture Construction*. WICSA, 2008.

[8] LTSA. Labeled Transition System Analyzer, http://www.doc.ic.ac.uk/ltsa/