Modeling context-dependent aspect interference using default logics

Frans Sanen, Eddy Truyen, and Wouter Joosen

Distrinet Research Group, Department of Computer Science, K. U. Leuven,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{frans.sanen, eddy.truyen, wouter.joosen}@cs.kuleuven.be

Abstract. Explicitly representing aspect interactions is vital so that they can be shared and used in the course of system evolution. As a consequence, guidance can be given to the software developer and automated support for handling interactions becomes possible. In this paper, we propose to use default logics for modeling context-dependent aspect interference. We motivate and illustrate our work by an example interference from the domotica world.

Keywords: Aspect interactions, knowledge, interference, default logics.

1 Introduction

Aspect interference is a well-known challenging problem with current aspect-oriented programming technology. As it has already been motivated in [10], explicitly representing aspect interactions results in an important form of knowledge that can be shared and used in the course of system evolution. If specified formally enough, software systems can exploit this knowledge to autonomously reconfigure themselves to detect and resolve undesired aspect interferences, by using existing safe dynamic reconfiguration support similar to the one in [13].

In this paper, we want to make the case for modeling support for context-dependent interferences. We define aspect interference as a conflicting situation where one aspect that works correctly in isolation does not work correctly anymore when it is composed with other aspects. A context-dependent interference is an interaction that might or might not occur if certain aspects are composed depending on the runtime context at hand. Or more formally: "Given an aspect A that is woven into a system S, there exists a set of contextual conditions C_A associated with aspect A such that, when at least one element of C_A evaluates to true, the execution of the aspect A will cause an error in the execution of system S. A contextual condition is defined as a boolean expression that evaluates over properties of the context in which the aspect is deployed – contextual properties." Obviously, the context of aspect A does not only consist of the system S but also involves all the other aspects that are simultaneously woven into S. As a consequence, context information entails key information pieces that we need to express.

We consider this particular problem of context-dependent aspect interferences in the case of aspect-oriented middleware [14, 7, 13] which uses AOP for implementing middleware services. Subtle aspect interferences exist in a middleware environment. Consider the example of a power saving aspect and an integrity aspect using symmetric encryption [11]. A symmetric encryption key has a limited lifetime and therefore should be regenerated upon expiration, which is very computationally

intensive. Only when the power of the device being used is low and the key is about to expire, interference arises between both the power saving and integrity aspect.

A prerequisite for the scenario of systems capable of autonomously reconfiguring themselves to resolve context-dependent interferences is that interaction knowledge has to be specified in an unambiguous way. We have found no satisfactory solutions in current work on interaction modeling. We will elaborate on this later in the paper.

The rest of this paper is structured as follows. Section 2 elaborates on the need for modeling context-dependent interactions. It also shortly indicates that current approaches lack sufficient support in this regard. We propose to use default logics for modeling context-dependent interactions in Section 3 before concluding in Section 4.

2 Modeling aspect interactions

To be able to share and use aspect interactions in the course of system evolution, we need a means for modeling them. Some work already exists where interactions are modeled separately, but to the best of our knowledge, these suffer from several shortcomings, especially in the context of context-dependent interactions. In the NFR framework [2], Chung et al. introduce the concept of correlating (i.e. interacting) non-functional requirements. It for instance can be expressed that using a compressed format to store information deteriorates (*hurts*) its response time. However, such a representation cannot take into account the concrete context in which the interaction arises, e.g. when the CPU load is above a certain threshold. Similarly, interaction modeling in feature models [4, 6] allows you to express that feature *A* requires or excludes feature *B*, but this is not flexible enough to provide any means to model the context on which an interaction depends. Classen et al. [3] consider feature interactions as the simultaneous presence of several features causing malfunctions, hence ignoring the potential context dependence of an interaction. Finally, Pawlak et al. [8] propose a language to abstractly define an execution domain, advice codes and their often implicit execution constraints. Especially the latter are relevant because exactly these represent the context in which undesired effects occur, e.g. a network overload situation. These conditions are key information pieces we need to express.

The pedagogical example interaction we will use throughout the rest of this paper is situated in a home integration system product line context and borrowed from [5]. Home integration systems are a new and emerging set of systems combining features in the area of home control, home security, communications, personal information, health, etc. Each feature easily can be mapped to one or more aspects implementing it. Imagine a domotica product that helps to protect the housing environment. On the one hand, your personal product entails a flood control feature which shuts off the water main to the home during a flood. On the other hand, it also contains a fire control feature that turns on some sprinklers during a fire. Turning the sprinklers on during a fire and flooding the basement before the fire is under control results in a really undesirable interaction since the flood control feature will shut off the home's water main, rendering the sprinklers useless. As a result, your house further will burn down.

In order to have a correct representation for our example interaction, three scenarios have to be considered: (1) the basement is flooded, (2) a fire in the house is detected and (3) the basement is flooded as a result of the sprinklers trying to extinguish the fire.

Traditional methods and technologies often offer support to prioritize features in relationship with one another. However, we are convinced that such a prioritization not always is feasible to overcome context-dependent interactions. One of the main reasons is because priorities are far less flexible. First of all, an interaction between two features having the same priority cannot be resolved. Secondly, the priority of two features related to one another can be different in varying circumstances. For instance, suppose there are two additional features included in your domotica product: a presence simulation feature that turns lights on and off to simulate the presence of the house occupants and a doorkeeper feature which controls the access to the house and allows occupants to talk to the visitor [12]. Obviously, we would like the doorkeeper not to give away the fact that nobody is at home if there is an unidentified person in front of the door to prevent the owners from a burglary.

3 Using default logics

Default logics haven been originally proposed by Reiter [9] as a non-monotonic logic to formalize reasoning with default assumptions. It allows us to make plausible conjectures when faced with incomplete information and draw conclusions based upon assumptions. [1] As an intuitive example of what can be expressed, consider the well-known principle of justice in our Western culture: "In the absence of evidence to the contrary, assume that the accused is innocent." In this section, we shortly will overview both the syntactic sugar and semantics (informally) of default logics by applying it to our example interaction from above. Next, we discuss the relevance of using default logics in our example.

3.1 Syntax and semantics

A default theory T is a pair (W, D) consisting of a set W of predicate logic formulas (background theory or *facts* of T) and a set D of defaults. The default explicitly representing our example interaction is presented below (1) and should be thought of being used together with the classical rule that is also shown (2).

$$\frac{waterInBasement: \neg active(fireControl)}{active(floodControl)} \quad (1)$$

$$fireDetected \rightarrow active(fireControl) \quad (2)$$

According to default (1), if we know that $waterInBasement$ is true and $\neg active(fireControl)$ can be assumed, we can conclude $active(floodControl)$. Because of rule (2), $active(fireControl)$ will be concluded upon fire detection.

The three parts of a default rule are called the *prerequisite* φ, *justifications* ψ_i and *conclusion* χ respectively. Hence, the general explanation of any default rule is given by "if we believe that *prerequisite* is true, and the *justification* is consistent with our current beliefs, we also believe the *conclusion*". In other words, given a default $\varphi: \psi_1, \psi_2, \ldots / \chi$, its informal meaning is: if φ is known, and if it is consistent to assume ψ_1,

ψ_2, … then conclude χ. It is consistent to assume ψ_i iff the negation of ψ_i is not part of the background theory *W*.

At this point, it is important to realize that classical logic is not appropriate to model this situation. Imagine the following rule as an alternative for (1).

$$waterInBasement \wedge \neg active(fireControl) \rightarrow active(floodControl) \qquad (3)$$

The problem with this rule is that we have to definitely establish (basically because of the closed world assumption) that the fire control feature is not active before applying this rule. As a consequence, the flood control service never would be able to become active.

The semantics of default logic typically is defined in terms of extensions. Intuitively, an extension seeks to extend the set of known facts (i.e. background theory) with "reasonable" conjectures based on the applicable defaults. More formally, a default φ: ψ_1, ψ_2, … / χ, is applicable to a deductively closed set of formulas *E* iff $\varphi \in E$ and $\neg\psi_1 \notin E$, $\neg\psi_2 \notin E$, ... You can think of *E* as the context in which φ should be known and with which ψ_i should be consistent.

3.2 Discussion

We will now revisit our default (1) together with its semantics. Intuitively, this rule states that the flood control service will be activated upon detection of water in the basement, unless the fire control feature is active. It is easy to see that with this representation all possible scenarios are represented correctly. In each of these scenarios, the set *D* of defaults contains default (1). The only two facts that are relevant when searching extensions are $waterInBasement$ and $fireDetected$.

If, on the one hand, a sensor detects water in the basement, then the background theory *W* will include $waterInBasement$. Because of default (1), the only valid extension is the one where flood control service will become active (we conclude $active(floodControl)$ because $waterInBasement$ (the prerequisite) is true and the justification $\neg active(fireControl)$ is not inconsistent with what is currently known.

On the other hand, if a fire is detected by the system, *W* will include $fireDetected$ and classical rule (2) fires so that $active(fireControl)$ also becomes true in the extension. If later (the third scenario), as a consequence, the basement will be flooded, default (1) can no longer be applied. Note that this is exactly what we wanted.

In our approach, the context in which an interaction occurs is made explicit via one or more justifications in a default rule. By taking certain conditions into account, the solution of the interaction lies in the fact that the justifications need to be invalidated in order to have a correct functioning system. Because of this, an interaction is prevented from occurring while normal execution behavior also easily can be captured and isn't influenced.

4 Conclusion

To conclude, we started from the observation that modeling aspect interactions results in an important form of knowledge that can be shared and used in the course of

system evolution. We propose to use default logics for representing aspect interactions. The main advantage of this approach is that the interaction becomes explicit in the justification part of a default rule. Therefore, undesired interactions can be prevented from happening by invalidating one of the justifications of the default rule representing the interaction.

Acknowledgments. This work is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, Research Fund K. U. Leuven and European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008.

References

1. Antoniou, G.: A tutorial on default logics. ACM Computing Surveys 31 (4), pp. 337-359, 1999.
2. Chung, L., Nixon, B. A., Yu, E., Mylopoulos, J.: Non-functional requirements in software engineering. Kluwer academic publishing, Norwell, 2000.
3. Classen, A., Heymans, P., Schobbens, P.: What's in a Feature: A Requirements Engineering Perspective. Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE'08), pp. 16-30, 2008.
4. Czarnecki, K., Eisenecker, U. W.: Generative Programming. Addison Wesley, London, 2000.
5. Kang, K.C., Lee, J., Donohoe, P.: Feature-oriented product line engineering. IEEE Software, vol. 19, no. 4, pp. 58-65, 2002.
6. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical report CMU/SEI-90-TR-021.
7. Lagaisse, B., Joosen, W.: True and transparent distributed composition of aspect components. 7th International Middleware Conference, pp. 41-62, 2006.
8. Pawlak, R., Duchien, L., Seinturier, L.: CompAr: Ensuring safe around advice composition. 7th Internation Conference on Formal Methods for Open Object-Based Distributed Systems, 2008.
9. Reiter, R.: A logic for default reasoning. Artificial Intelligence 13 (1-2), pp. 81-132, 1980.
10. Sanen, F., Truyen, E., Joosen, W.: Managing concern interactions in middleware. 7th International Conference on Distributed Applications and Interoperable Systems, 2007.
11. Sanen, F., Truyen, E., Joosen, W., Jackson, A., Nedos, A., Clarke, S., Loughran, N., Rashid, A.: Classifying and documenting aspect interactions. Proceedings of the 5th AOSD Workshop on Aspect, Components, and Patterns for Infrastructure Software, pp. 23-26, 2006.
12. Schwanninger, C. et al.: Confidential list of requirements on a Totally Integrated Home platform. Siemens internal document, 2006.
13. Truyen, E., Janssens, N., Sanen, F., Joosen, W.: Support for Distributed Adaptations in Aspect-Oriented Middleware, 7th International Conference on Aspect-Oriented Software Development, 2008.
14. Truyen, E., Vanhaute, B., Joosen, W., Verbaeten, P., Jorgensen, B.: Dynamic and Selective Combination of Extensions in Component-based Applications. 23rd International Conference on Software Engineering, pp. 233-242, 2001.