State-Based Join-Points: Motivation and Requirements

[Position paper]

Nelis Boucké
Tom Holvoet
AgentWise, DistriNet, K.U.Leuven
Celestijnenlaan 200A,
B-3001, Leuven, Belgium
{nelis.boucke,tom.holvoet}@cs.kuleuven.ac.be

ABSTRACT

In developing a real-world complex application, we experience the major problem that complex concerns do not easily map onto low-level aspects with join-points based on fixed points in the program code. It is our observation that modularizing concerns and quantification are to be tackled at design-time, using suitable abstractions, with a translation to dynamic weaving at run-time. In particular, we argue that 'abstract states' of software entities (a concern consists of software entities) are a promising instrument for defining higher-level join-points for concerns. The specification and quantification of concerns in terms of abstract states typically result in dynamic weaving, i.e. depending on run-time states of the software entities.

Based on experience, we provide requirements for supporting concern modelling and quantification at design-time, as well as an initial sketch of an approach that we investigate in this perspective. The approach is based on a new type of higher-level join-points, called *state-based join-points* and serves as an example of the necessity for advanced dynamic (state-based) aspect weaving. The approach is motivated and illustrated through a scenario in a real-world application, namely decentralized control software for several automatic guided vehicles in an industrial transportation system.

Categories and Subject Descriptors

D.2.11, D.2.2 [**Software Engineering**]: Software Architectures, Design Tools and Techniques

General Terms

Design

1. INTRODUCTION

Research in Aspect Oriented Software Development (AOSD) focusses on developing systems with several non-orthogonal (e.g. overlapping, crosscutting and interacting) concerns. In the AOSD community great work has been done in building the technology to support separation of concerns, both statically [10, 13, 9, 11] and dynamically [1, 2].

In this position paper we argue that there is more to dynamic aspect approaches than technologies to support variety of implementation code join points. In particular, a software designer should be able to model concerns and explicitly describe quantification [6], which results in dynamic aspect weaving at run-time. To this end, implementation code join points do not provide an appropriate level of abstraction. Here, state-based join-points and quantification are defined, describing concern interference on a higher abstraction level.

We base the arguments and the proposal on our experience in developing a complex decentralized application for controlling automatic guided vehicles (AGVs) in a warehouse management system. For obvious reasons, we aim to apply the principles of separation of concerns throughout the design of the application. For example in the architectural design phase interesting concerns can be identified. However, these concerns do not easily map onto low-level aspects and join-points based on fixed points in the program code. Such a concerns are typically made up by a set of related software entities (e.g. objects or components). We observe that suitable quantification of these concerns can be achieved by considering 'abstract states' of software entities, rather than fixed points in program execution. Such an abstract state are more than the value of a attribute, it is relevant state on a higher abstraction level with a clear semantic meaning, described in terms of operations calls and attributes of software entities. A corresponding new type of higher-level join-points is introduced, called *state-based join-points*. The need for a dynamic join-point model is supported by publications on events in AOP ([7, 4]), the importance of a clear semantic meaning came forward in [15]. High-level quantification based on state-based join-points obviously results in a highly dynamic run-time system, requiring dynamic weaving.

First, in sec. 2, the necessity of higher-level concern quantification is motivated by pointing out a concrete problem scenario that we encounter in a real-world application of AGVs. Next, in sec. 3, an initial list of requirements and an initial sketch for design-level concern modelling and quan-

tification is presented, based on our experience with this application. Finally, in sec. 4, we conclude with a number of open issues and links towards future work.

2. REAL-WORLD APPLICATION: AUTOMATIC GUIDED VEHICLES

This section contains a description of the real-world application, the identification of some important concerns in the architecture and a more detailed problem statement.

2.1 Application

Currently, our group is active in a research project with an industrial partner to decentralize control in this transportation system using a multi-agent system (EMC^2 [5]). An AGV is an unmanned, computer-controlled transportation vehicle using a battery as energy source. AGVs are able to perform transportation tasks, consisting of picking up a load and transporting it to the destination. High-level functional requirements for the AGV system are: (1) allocating transportation tasks to individual AGVs; (2) performing those tasks; (3) preventing conflicts between AGVs on crossroads; and (4) charging the batteries of AGVs on time.

Multi-agent systems (MASs) provide an approach for solving software problems by decomposing a system into a number of autonomous entities (e.g. AGVs), embedded in an environment, which cooperate in order to reach the functional and non-functional requirements of the system [14]. In general, MASs form a family of software architectures [16].

2.2 Architectural concerns

The need for state-based join-points is identified during architectural design of an AGV system. Once the basic structure of the architecture is clear, there are several concerns (specific to this architecture) that are important for the designer of the system. The basic architectural structure used here is a MAS, decomposing the system in agents (AGVs) and an environment (the factory). In essence, a concern is 'an issue that is important for a stakeholder in the system'. Concerns specific to the basic structure of an architecture are denoted with the term *'architectural concerns'* and are 'issues, part of a specific basic architecture, that are important for a designer using this architecture'. A few architectural concerns have been listed here to illustrate our position:

Autonomy One of the fundamentals of an agent is that it operates autonomously and proactively. E.g. an AGV-agents decides for itself what it will do, e.g. perform a task, reload its battery. Autonomy is an architectural concern because the way it is filled in has an important influence on the remainder of the architecture.

Individual capabilities Each agent has a collection of capabilities it can perform. Both externally visible actions (like driving, picking or dropping down a load) and internal calculation (like calculating the shortest path) belong to this concern. The capabilities are an important factor in determining the internal structure of an agent, asking for attention on an architectural level.

Coordination A MAS is built using several agents and these agents have to coordinate to meet the overall goals. For example in the AGV system, agents coordinate their behavior to prevent collisions on crossroads. Coordination of entities is a non-trivial problem that asks for an architectural solution. It is not our intent to focus on coordination itself, but rather on how coordination can be combined with other concerns.

Clearly, these concerns cross-cut and overlap each other. For example, consider the concerns autonomy and individual capabilities. Autonomy has an important influence on how individual capabilities are filled in. Both concerns overlap and crosscut. From the designer point of view, it is good to keep these concerns separated throughout the remaining building process, following the philosophy of SoC.

2.3 Problem statement

A simple scenario of the AGV system is used to illustrate our position. Consider two AGVs approaching a crossroad, leading to a possible collision between them. To enforce that only one AGV at a time can enter the crossroad, coordinating the behavior of both AGVs is needed. From the AGV system designer's point of view, there are several Architectural concerns involved, we consider 'individual capabilities' and 'coordination' as an example. If an AGV is approaching a crossroad (with 'approaching crossroad' as abstract state) and if there is another AGV approaching this crossroad leading to a possible conflicting situation (with 'other approaching' as abstract state), the concern 'coordination' crosscuts the concern 'individual capabilities'.

It is very hard, if not impossible, to specify quantification of these concerns based on fixed points in the program code of software entities. Simply because the quantification is not dependent on an operation call or attribute value, but rather on an abstract state (e.g. 'approaching crossroad'). As stated before, such an abstract state is more then a value of one or two attributes, it is relevant state on a logical level with a clear semantic meaning, described in terms of operations calls and attributes of software entities. E.g. to determine if an AGV is approaching a crossroad, the position of the AGV and its driving direction are compared with an internal layout of the factory. This type of behavior can not easily be expressed with current dynamic aspect approaches using the classical type of join-point, leading towards a design where architectural concerns are not clearly separated.

3. STATE-BASED JOIN-POINTS

It is our position that concern based approaches must support a level of abstraction that is higher than implementation code join-points. They must go beyond changing behavior based on the call-stack context, co-occurrence of predicate triggers or concurrent thread status (CFP of this workshop [3]), moving towards a higher level of abstraction.

3.1 Requirements for state-based join-points

An initial list of requirements for a mechanism based on state-based join-points is presented here, making up our research challenges for the next few years.

First, it must be possible to **define** a state-based join-point (in addition to implementation code join-points). For this definition, a rich language is needed to determine the abstract state.

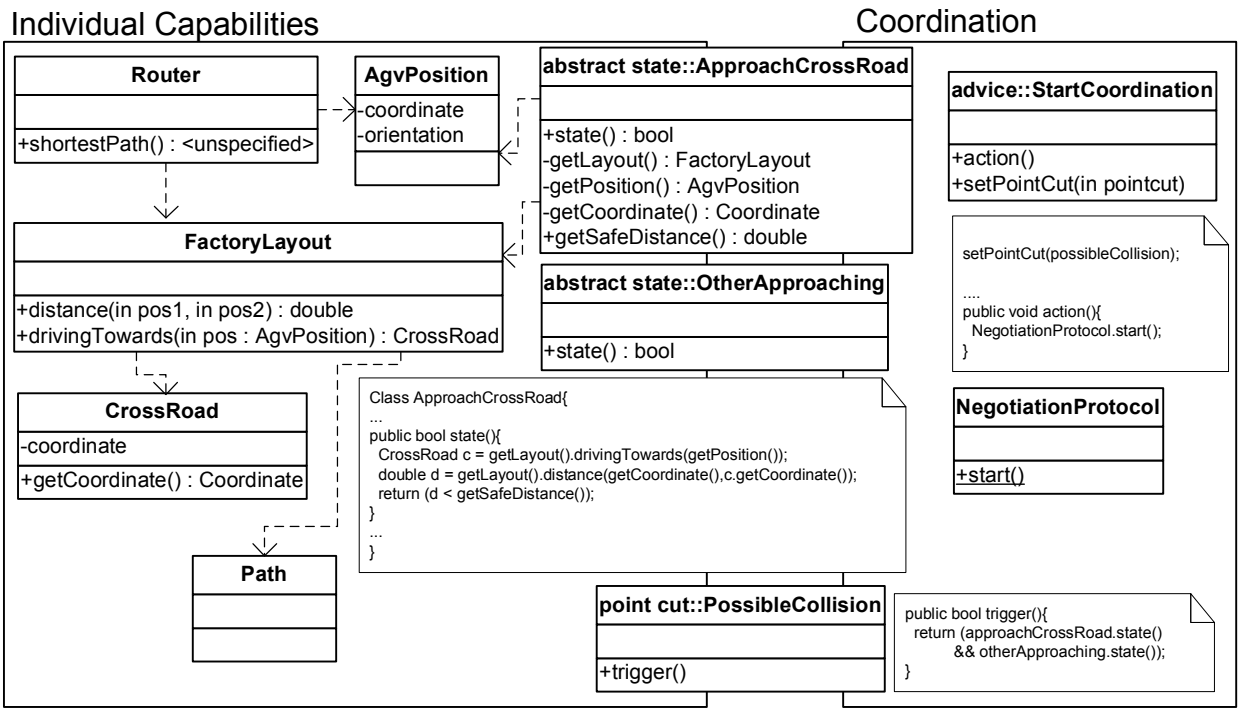

Figure 1: A graphical representation of the Individual Capabilities concern.

A second requirement is that a **declarative quantification** language is needed, backing away from enumeration or pattern matching based on names [8, 12].

A third requirement is that a state-based join-point mechanism must respect the **obliviousness property** [6]. The obliviousness property states that the development of a software entity must be possible without being aware of the aspects that eventually will crosscut it. This implies that it must be possible to (1) define state-based join-points separately from the involved software entity; (2) after it is fully developed; and (3) without enhancements to it.

The fourth requirement is that both the definition of state-based join-points and the quantification of these join-points must be **translatable to a concrete execution** of a program. Thus, there is a need for a dynamic weaver, capable of working with state-based join-points.

The fifth requirement is **prevention of semantic mismatch conflicts**. Semantic mismatch conflicts between concerns are a typical problem in AOSD approaches [15]. Because every state-based join-point needs a clear semantic meaning, prevention of semantic mismatches is an important requirement.

3.2 Initial sketch

In this section we present a preliminary attempt towards supporting state-based join-points. The approach consists of three steps: seperation/modularization, defining state-based join-points and quantification. In this initial sketch, the approach is presented as a UML-diagram.

First, architectural concerns must be separated and modularized. Although modularization is an important issue, we only briefly describe it because our focus is on state-based join-points. In this initial sketch an UML-diagram corresponds with a concern and the software entities making up the concern are classes and objects. In Figure 1 an example is depicted, containing a few classes of the concern 'individual capabilities' and 'coordination' to illustrate our approach.

Next, the state-based join-points must be defined. First, abstract states are defined (e.g. `ApproachCrossRoad` and `OtherApproaching`). The `state` operation represents the condition for this abstract state. Next, a pointcut is defined, based on these abstract states (e.g. `PossibleCollision`). The `trigger` operation contains a condition when the quantification should be triggered (enumerated here, the final system should use a declarative description). The variables `approachCrossRoad` and `otherApproaching` represent the instantiations of the abstract states 'approaching crossroad' and 'other approaching' respectively. Every pointcut is associated with a particular concern and thus the concern definition is extended with a new, externally visible, join-point.

Finally, quantification is specified using state-based join-points. For this, we write an advice for a particular pointcut (e.g. `StartCoordination`). The `setPointCut` operation sets the pointcut for this advice, the `action` operation contains the action for this advice.

The results of these three steps serve as input for the dynamic weaver. Run-time weaving based on state-based join-points will probably be one of the most challenging problems for our future research. Especially, determining when state checks must be performed and thus where concerns should

be woven is one of the main problems. Using current aspect technology could be useful in this step. Until know it is unclear what the exact implications are of run-time weaving and state checks. But it is certain that the amount of state checks and the checks itself will have an influence on the performance of the system.

Remark that using the terminology of [6], our initial sketch of state-based join-points is a event-based, publish and subscribe (EBPS), black-box AOP-mechanism. Differences with classical EBPS systems are that both the events (state-based join-points) and the subscription (quantification statements) are described separately. The main differences with quantification of events (in [7]) and Event-based AOP (EAOP [4]) are that our "events" are: (1) high-level state changes with a clear meaning, in contrast with primitive events, similar to fixed points in the program execution; and (2) defined explicitly and separately. In addition, [4] differs in its aspect definition (an aspect is a transformation on an event) and the way the weaving takes place (using a central execution monitor, catching every event and applying every aspect).

4. CONCLUSION

In this paper we outlined our position that dynamic aspect approaches must support an abstraction level that is higher than implementation code join-points and illustrated this in an industrial transportation system. High-level quantification based on state-based join-points has been introduced. This obviously results in a highly dynamic run-time system, requiring dynamic weaving.

What the exact implications of using state-based join-points are and how the join-points will translate to dynamic weaving are still important open issues. Especially, where exactly the weaving should take place and which current technology for dynamic aspect can be (re)used for defining dynamic weaving based on state-based join-points are important research tracks for the near future. A crucial step will be to determine a set of requirements for the dynamic weaver working with state based join-points and to consider the possible trade-offs.

5. ACKNOWLEDGMENTS

Thanks to the AgentWise taskforce, the EMC^2 members and the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

6. REFERENCES

[1] Lodewijk Bergmans and Mehmet Aksit. Principles and design rationale of composition filters. In R. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect-Oriented Software Development.* Addison-Wesley, 2004. ISBN 0-32-121976-.

[2] I. Sommerville Chitchyan, R. Comparing dynamic ao systems. *Dynamic Aspects Workshop (at AOSD04).*, pages 23–36, 2004.

[3] DAW Organizing Committee. Call for papers of the dynamic aspects workshop (daw05). `http://aosd.net/2005/workshops/daw/cfp.html`. Checked on 13 Januari 2005.

[4] Rémi Douence and Mario Südholt. A model and a tool for event-based aspect-oriented programming (eaop). Technical report, Ecole des Mines de Nantes, 2002.

[5] Egemin and DistriNet. Emc^2: Egemin modular controls concept. IWT-funded project. From 1 March 2004, until 28 February 2006.

[6] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of the workshop on Advanced Separation of Concerns, OOPSLA*, 2000.

[7] R. E. Filman and K. Havelund. Source-code instrumentation and quantification of events. In *Foundations of Aspect-Oriented Languages (FOAL'02)*, 2002.

[8] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69. ACM Press, 2003.

[9] IBM. Concern manipulation environment (CME). `http://www.research.ibm.com/cme/`. Checked on 4 januari 2005.

[10] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. *Proceedings European Conference on Object-Oriented Programming, Springer-Verslag*, 1241:220–242, 1997.

[11] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-oriented programming with adaptive methods. *Commun. ACM*, 44(10):39–41, 2001.

[12] István Nagy, Lodewijk Bergmans, and Mehmet Aksit. Declarative aspect composition. `http://trese.cs.utwente.nl/publications/files/0201Nagy.pdf`, 2004. Checked 13 januari 2005.

[13] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2000.

[14] H. V. D. Parunak. Agents in overalls: Experiences and issues in the development and deployment of industrial agent-based systems. *International Journal of Cooperative Information Systems*, 9(3):209–227, 2000.

[15] Peri Tarr, Maja D'Hondt, Lodewijk Bergmans, and Cristina Videira Lopes. Workshop on aspects and dimensions of concern: Requirements on, and challenge problems for, advanced separation of concerns. *Lecture Notes in Computer Science*, 1964:203–243, 2001.

[16] D. Weyns, A. Helleboogh, E. Steegmans, T. De Wolf, K. Mertens, N. Boucke, and T. Holvoet. Agents are not part of the problem, agents can solve the problem. In *Proceedings of the OOPSLA 2004 Workshop on Agent-oriented Methodologies*, 2004.