Behavioral Types for Component-based Development of Cyber-Physical Systems

Jan Olaf Blech[1] and Peter Herrmann[2]

[1] RMIT University, Melbourne, Australia, janolaf.blech@rmit.edu.au
[2] NTNU, Trondheim, Norway, herrmann@item.ntnu.no

Abstract. Spatial behavioral types encode information on the tempo-spatial behavior of components acting in the physical space. That makes it possible to utilize the well established concept of type systems with its well studied benefits for programming languages, e.g., fast automatic detection of incompatibilities and coercion, also in the cyber-physical world of domains such as embedded systems. So, spatial behavioral types support development and better maintenance of systems leading to a reduction of errors, improvement of safety and, in consequence, lower expenditure. In this position paper, we summarize existing work and develop our ideas for a spatial behavioral type concept. In particular, we turn our attention to making the spatial behavioral types easily usable by non-experts. Besides of a semantics that resembles traditional types systems, our method offers a syntax based on easily comprehensible regular expressions while systems can be verified using fully-automatic tools.

1 Introduction

Most programming languages use type systems that facilitate the automatic analysis of program code for errors. Behavioral types [3, 4] are an enhancement of this well-known concept. In contrast to simple type systems, they do not only model interfaces on a purely syntactical level but also take the behavior of software into consideration (see [8]). Behavioral types support Human Factors in two ways:

– They provide an easily comprehensible modeling language for abstract component specifications. This can rely on formal specification mechanisms such as regular expressions that are frequently used by non-experts.
– In the context of developing component-based software, behavioral types provide means to specify and check component contracts that consider the current states of a component and its environment. This makes an easy and mostly fully automatic analysis of the conformance of a component with its environment possible (see, e.g., [14, 32]). The checks may run in the background of development environments or deployed systems and interact with user-interfaces in an intuitive way.

In the approach presented here, we bring the spatial behavior of cyber-physical components into the type system world. In domains such as the automotive industry and industrial automation, standards like ISO 26262 and IEC

61499 gain increasing popularity. Since several of these standards support a component view, we believe that the behavioral types concept can play an important role in supporting the development, maintenance and service activities of the components in a supportive and user-friendly way.

We continue with a summary of related work in Sect. 2. To ease the understanding of our approach, we introduce a motivating example in Sect. 3 followed by a description of the core concepts of the behavioral types in Sect. 4. Thereafter, we discuss the particular properties of the spatial behavioral types in Sect. 5. The text is completed by some concluding remarks.

2 Existing Approaches

Different behavioral type-like approaches to specify interfaces of component systems and reason about these specifications have been proposed in the past. The current state-of-the-art, however, focusses almost exclusively on software aspects.

Interface automata [3] are one form of behavioral types. Component descriptions are based on timed automata. The focus of interface automata is on communication protocols between components. Interface automata do not target all type relevant aspects discussed in this paper, e.g., physical and spatial aspects or the checking the behavior at runtime of a component by using some form of monitoring. The main focus is an compatibility checks of software components interacting at compile time. Behavioral types are part of the Ptolemy framework [35]. Here, one focus is on the software part of real-time systems such as execution time of code.

The idea of having well defined specifications defining interfaces of software component systems has been made popular by design-by-contract [36] like approaches during the late 80s and early 90s. The focus of the classical approach is on contracts for object oriented systems. Other work that is related to our behavioral types comprises specification and contract languages for component based systems that have been studied in the context of web services. For example, the approach presented in [2] comprises request and response operations as a means for specifying behavior. Process algebra-like approaches including deductive techniques are presented for web services [18, 20]. In [18], emphasis of the formalism is put on compliance, a correctness guaranty for properties like deadlock and livelock freedom. Another algebraic approach to service composition is described in [23]. Means restricting the interface behavior of OSGi for facilitating analysis are featured in [16].

A variant is used in the model-based system engineering technique SPACE [34] and its tool-set Reactive Blocks. Using UML activities, systems are modeled by composing descriptions of subfunctions that are arranged in so-called building blocks. A building block is provided by an External State Machine (ESM) [32] that is a UML state machine describing the interface behavior of the building block. Due to formal semantics of the UML activities and state machines [33], one can verify by model checking at design time that a building block complies with both its own ESM and those of the blocks it incorporates in its behavioral description. The approach has already been used in the context of cyber-physical systems [27, 30].

JML [22] can be applied to specify pre- and postconditions for Java programs. Although not a type system, it can be utilized to specify aspects of behavior for Java based systems. In addition, assertion like behavioral specifications have been studied for access permissions [21]. Behavioral types comprising behavioral checks at runtime for component based systems were proposed in [4]. The focus is on the definition of a suitable formal representation expressing types and investigating their methodical application in the context of a model-based development process. A language for behavioral specification of software components, in particular of object oriented systems, is introduced in [31]. Compared to the more requirement-based descriptions proposed in our paper, the specifications used in [31] are still relatively close to an implementation. Additional work on refinement of automata based specifications is studied in [38]. A survey with a focus on pre-/postcondition and invariant-based annotations for programming languages can be found in [28].

The runtime verification community has developed frameworks which can be used to generate monitors checking behavioral type conformance at runtime in order to detect and report type violations. The MOP framework [37] provides the integration of specifications into Java source code files and generates AspectJ aspects realizing runtime monitoring. A framework that regards the trade-off between checking specifications at runtime and at development time is provided in [17]. The framework described in [6] facilitates also the generation of Java monitors but leaves the instrumentation aspect, i.e., the connection of the monitor to the deployed system, to the implementation. Other topics explored in this context comprise the efficiency and expressiveness of monitoring [5, 7]. Monitoring of performance and availability attributes of Java/OSGi-based systems has been studied in [41]. A focus is on the dynamic reconfiguration ability of OSGi. Work building on the .Net framework for runtime monitor integration is described in [26]. Runtime monitors for interface specifications of web-service in the context of a concrete e-commerce service are described in [25]. Behavioral conformance of web-services and corresponding runtime verification were investigated in [19]. Furthermore, in [24] runtime monitoring of web-services is studied, in which runtime monitors are derived from UML diagrams. Runtime enforcement of safety properties is especially important in the context of cyber-physical systems, since a deviation is not only reported, but a countermeasure is provided. In [39], security automata are used enforce security properties. These automata are able to halt the underlying program when a deviation from the expected behaviors is detected. A similar approach to protect software components against malicious behavior, is provided in [29]. Behavioral types-based runtime monitoring for industrial automation is also studied in [44], where the abstract behavioral types specification yields a monitor that runs directly on PLC.

Tangible user interfaces [40] provide a good way of exemplifying underlying principles of cyber-physical behavioral types. Here, each component in the

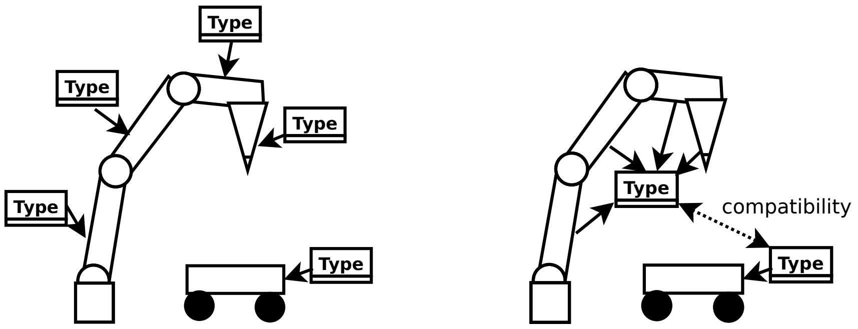

Fig. 1. Behavioral types for a robot

interface may bear cyber-physical characteristics and can be described with a behavioral specification, e.g., using a domain specific language.

3 Motivating Example

Figure 1 shows the interaction of a robot with a vehicle. Both components may change their positions, directions and orientations over time. We can represent the spatial attributes of a component as subtypes of its spatial behavioral type. For instance, the position of the vehicle at a certain point of time can be expressed by the space it physically occupies. The space may be represented by coordinates in a geometric coordinate system stored in the behavioral type. Using theses subtypes, one can verify spatial properties, e.g., two components do not collide if their physically occupied spaces never overlap at any point of time.

Cyber-physical components can be made up of other components. For example, the robot consists of three segments and a tool that are attached via joint devices. Each subcomponent has its own spatial behavior that can be represented by a separate spatial behavioral type. The type encodes the movement of its sub-component which depends on the spatial behavior of the attached devices. Similar subcomponents like the lower two segments of the robot may be represented by the same types. The two type instances relate to different attached segments which leads to different positions and orientations.

The picture on the right side of Fig. 1 depicts the composition of the behavioral types of the three segments and the tool to a single type that superimposes the spatial movements of the subcomponents and represents the spatial behavior of the overall robot. The composition makes it easier to check if the spatial behaviors of the robot and vehicle are compatible, e.g., that the vehicle is in a certain distance when the robot loads something onto it.

The concept facilitates also the reconstruction of cyber-physical components. For instance, when a segment of the robot is replaced by another one, only its local behavioral type has to be exchanged. If the type of the replacing component is a refinement of the one of the replaced component, one can assure certain spatial properties without demanding a new proof. E.g., if we could prove that a certain vehicle cannot collide with the robot since it never passes an area reachable by the robot arm, this property will also hold if a robot segment is replaced by a shorter one.

4 Core Concepts of Behavioral Types

We present some core concepts on behavioral types to support a development process of component based systems. To ease their application for users familiar with traditional *type systems*, (spatial) behavioral types should provide a number of features [14]:

- **Abstraction** Behavioral types represent aspects of programs, components, or systems. They provide an *abstraction from details concerning interaction with their environment as well as their internal structure* [14]. For example, the driver of a physical component may require that the interaction with its environment follows a particular protocol that may form a part of the abstract view of the component provided by its behavioral type. According to the classification in [8], abstraction is a typical example of *behavioral contracts* that one can find, e.g., in UML and programming languages like Eiffel. It is also alike to the ESMs used in SPACE and Reactive Blocks [32] (see Sect. 2).
- **Type conformance:** Type conformance is used to relate a component to its own behavioral type. For instance, one can check whether a component interacting with its environment obeys the protocol listed in its behavioral type under all circumstances. These proofs can be taken at development time of a system, during changes of its component structure, or at runtime. Type conformance resembles classical static type checks performed by compilers of imperative programming languages.
- **Type refinement:** Behavioral types should support stepwise refinement, i.e., developing systems by making their models gradually more detailed using correctness preserving steps [1]. Thus, they have to be based on a formal semantics that allows to *ensure the correct implementation of abstract specifications by concrete components* [14]. This is quite similar to inheritance in object-oriented programming languages in which, as long as a program is developed "top-down", first more abstract functionality is created using super classes, while later on the functionality is refined by applying inherited classes.
- **Type compatibility:** To facilitate the composition of components to systems, one has to check whether a component fulfills not only its one behavioral type but also those of its environment in order to guarantee that the components interact with each other in the desired way. This can be investigated at development time and at runtime when dynamically reconfiguring a system. The development time-based analysis is alike static type checks while the dynamic tests resemble introspection in component-structured software (see [43]) as well as runtime monitoring.

- **Type inference:** As shown in the example description, subsystems can consist of various components. Often, type compatibility proofs can be easier if a subsystem is represented by a single behavioral type. Therefore, the formalism of behavioral types should *allow to infer the type of a composed component from the types of its constituents* [14]. Type inference corresponds to combining various software components to more comprehensive blocks resp. the composition of various building blocks to more extensive ones in Reactive Blocks [34].

To fulfill all these properties, the selected formalism has to take certain dependencies into consideration. For instance, type conformance has to guarantee with respect to type refinement that for a pair of components A and $\widehat{A}$ conforming to a pair of behavioral types T_A and $T_{\widehat{A}}$ with A subsuming the behavior of $\widehat{A}$, $T_{\widehat{A}}$ should be a refinement of T_A. Moreover, type refinement, type compatibility and type inference should agree that if a type T_A compatible to a given type T_B is refined by another type $T_{\widehat{A}}$, also $T_{\widehat{A}}$ should be compatible to T_B by definition. Similarly, if in a composed type T_S one type T_A is replaced by a refined type $T_{\widehat{A}}$ leading to a new composed type $T_{\widehat{S}}$, then $T_{\widehat{S}}$ should be a refinement of T_S. Also, for application in a development process, a behavioral type should not only be explicitly provided for a component and checked for conformance, but may be specifically constructed for this component. This is desirable in a seamless model-based development process. Finally, as type checking of expressive behavioral types is in general undecidable, an adequate level of expressiveness is needed making type checking feasible without over-restricting the expressiveness of the behavioral types.

We have studied behavioral types in the context of the OSGi framework [10–13]. Here, we use a simple finite automaton model (Σ, L, l_0, E) that consists of an alphabet Σ of labels, a set L of locations, an initial location l_0 and a set E of transitions. A transition is a tuple (l, σ, l') with $l, l' \in L$ and $\sigma \in \Sigma$. This formalism allows, e.g., runtime checking of type conformance [9]. Further, it is suitable to theorem proving as discussed in [14]. Nevertheless, the concept of behavioral types is suited to a diversity of formalisms. For instance, we currently experiment with temporal logic for cyber-physical systems (see [42]).

Behavioral types can be used for runtime-verification of systems, supplying a monitor being executed in parallel with a system implementation. The monitor corresponds to a behavioral type and checks all behavioral constraints specified via the type. It observes the system behavior and reports violations. The generation of the monitors from behavioral types can be performed automatically.

Furthermore, as already mentioned, the use of behavioral types facilitates the dynamic reconfiguration of systems based on type information and the discovery (both at runtime and development time) of components in a SOA like setting.

5 Spatial Behavioral Types

Spatial behavioral types extend the notion of behavioral types to cyber-physical components. Such components can comprise physical structures like the arms and tools attached to a robot as shown in the introductory example. Furthermore, controllers, network infrastructure elements, sensors and actuators are good candidates for spatial behavioral types.

Like purely software-related behavioral types, the spatial types comprise general aspects, e.g., protocols defining the interaction of a component with its environment. In addition, they define specific spatial aspects like the physical occupation of a physical components as well as its position, direction and speed. Depending on the application domain, further aspects as the acceleration of an object can be added. The representation of the spatial behavior is usually quite simple since we can restrict ourselves to describe positions by coordinates in the x, y and z axes in the Euclidian space.

An advantage of this proceeding is that, similar to software components, we do not need to model a physical unit with its full complexity from scratch. Instead, we can start with relatively abstract spatial behavioral types that are stepwise refined to more complex ones until we finally get one that considers all relevant spatial properties of the real component. This facilitates the handling of complexity in the development process vastly. Moreover, we can verify crucial spatial type compatibility properties, e.g., freedom of collisions or keeping a certain distance between the robot tool and the vehicle when a good is loaded or unloaded, based on the abstract models.

Of course, we have to guarantee that these proofs stay valid also for the refined models. For that, we use over- and underapproximation of physical properties. For example, in abstract models of a component, we can overapproximate the spatial area occupied by a component. That allows us to perform type compatibility proofs already with the coarse-grained models that will also hold for more detailed ones as long as the refined models do not exceed the overapproximated elongation of the original ones. An example for underapproximation is the assumption of low sensor ranges in abstract models. Thus, we can refine the sensor models reusing proofs for the original ones as long as their ranges is not shorter than those of the refinements.

Behavioral types are applicable for the specification of software controlled cyber-physical entities. Furthermore, they can be used to describe other entities such as overapproximations of human behavior, or elements in an environment where a cyber-physical system is deployed. Such elements can comprise static descriptions of obstacles such as walls and pillars in closed spaces or open-environment features such as lakes and mountains or infrastructures such as roads or rail-lines.

6 Conclusion

We motivated spatial behavioral types as an advanced concept for type systems. In the context of component-based system development, spatial behavioral types lift type systems to the component level also taking behavior of the underlying component into account. Thereby, they provide user-friendly means to specify and check contracts since easily comprehensible syntactical constructs are used, the core features of the method resemble techniques well-known from programming languages resp. component-based development, and the verification tools work automatically. In the past, realization of behavioral type systems for software was studied. In this publication, we propose the use of these type concepts also for components that comprise physical aspects which can be expressed using the concepts of Euclidian geometry.

Currently, we integrate spatial behavioral types into the model-based engineering technique SPACE [34] in order to facilitate the design of controllers for cyber-physical systems. Moreover, we work on the combination of spatial behavioral types with the verification tool-suite BeSpaceD [15] such that a highly automatic analysis of spatiotemporal properties will be possible. This will ease the application of the behavioral types for cyber-physical components further.

References

1. M. Abadi and L. Lamport. The Existence of Refinement Mappings. Theoretical Computer Science, 82(2):253–284, 1991.
2. L. Acciai, M. Boreale, and G. Zavattaro. Behavioural Contracts with Request-Response Operations. Science of Computer Programming, 78(2):248–267, 2013.
3. L. de Alfaro, T.A. Henzinger. Interface Automata. In Symposium on Foundations of Software Engineering, ACM, 2001.
4. F. Arbab. Abstract Behavior Types: A Foundation Model for Components and Their Composition. In Formal Methods for Components and Objects, vol. 2852 of LNCS, Springer-Verlag, 2003.
5. H. Barringer, Y. Falcone, K. Havelund, G. Reger, D. Rydeheard. Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In 18th International Symposium on Formal Methods (FM'12), vol. 7436 of LNCS, Springer-Verlag, 2012.
6. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In Verification, Model Checking, and Abstract Interpretation (VMCAI'04), vol. 2937 of LNCS, Springer-Verlag, 2004.
7. A. Bauer, M. Leucker. The Theory and Practice of SALT. In NASA Formal Methods, vol. 6617 of LNCS, Springer-Verlag, 2011.
8. A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. Computer, 32(7):38–45, 1999.
9. J. O. Blech. Ensuring OSGi Component Based Properties at Runtime with Behavioral Types. In 10th Workshop on Model Design, Verification and Validation Integrating Verification and Validation in MDE, 2013.
10. J. O. Blech. Towards a Formalization of the OSGi Component Framework. http://arxiv.org/abs/1208.2563v1. arXiv.org 2012.
11. J. O. Blech. Towards a Framework for Behavioral Specifications of OSGi Components. In 10th International Workshop on Formal Engineering Approaches to Software Components and Architectures. Electronic Proceedings in Theoretical Computer Science, 2013.
12. J. O. Blech, Y. Falcone, H. Rueß, B. Schätz. Behavioral Specification based Runtime Monitors for OSGi Services. In Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), vol. 7609 of LNCS, Springer-Verlag, 2012.

13. J. O. Blech, H. Rueß, B. Schätz. On Behavioral Types for OSGi: From Theory to Implementation. http://arxiv.org/abs/1306.6115. arXiv.org, 2013.
14. J. O. Blech and B. Schätz. Towards a Formal Foundation of Behavioral Types for UML State-Machines. In 5th International Workshop UML and Formal Methods. Paris, ACM SIGSOFT Software Engineering Notes, 2012.
15. J. O. Blech and H. Schmidt. Towards Modeling and Checking the Spatial and Interaction Behavior of Widely Distributed Systems. In Improving Systems and Software Engineering Conference, Melbourne, 2013.
16. S. Bliudze, A. Mavridou, R. Szymanek, A. Zolotukhina. Coordination of Software Components with BIP: Application to OSGi. In 6th International Workshop on Modeling in Software Engineering, ACM, 2014.
17. E. Bodden, L. Hendren. The Clara Framework for Hybrid Typestate Analysis. International Journal on Software Tools for Technology Transfer (STTT), 14:307–326, 2012.
18. M. Bravetti, G. Zavattaro. A Theory of Contracts for Strong Service Compliance. Mathematical Structures in Computer Science 19(3): 601–638, 2009.
19. T. D. Cao, T. T. Phan-Quang, P. Félix, R. Castanet. Automated Runtime Verification for Web Services. In International Conference on Web Services, IEEE Computer Society, 2010.
20. G. Castagna, N. Gesbert, L. Padovani. A Theory of Contracts for Web Services. ACM Trans. Program. Lang. Syst. 31(5), 2009.
21. N. Cataño and I Ahmed. Lightweight Verification of a Multi-Task Threaded Server: A Case Study With The Plural Tool. In Formal Methods for Industrial Critical Systems (FMICS), vol 6959 of LNCS, Springer, 2011.
22. P. Chalin, J.R. Kiniry, G.T. Leavens, E. Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In Formal Methods for Components and Objects (FMCO), vol. 4111 of LNCS, Springer 2005.
23. J. L. Fiadeiro, A. Lopes. Consistency of Service Composition. In Fundamental Approaches to Software Engineering (FASE), vol. 7212 of LNCS, Springer, 2012.
24. Y. Gan, M. Chechik, S. Nejati, J. Bennett, B. O'Farrell, J. Waterhouse. Runtime Monitoring of Web Service Conversations. In 2007 Conference of the Center for Advanced Studies on Collaborative Research, ACM 2007.
25. S. Hallé, T. Bultan, G. Hughes, M. Alkhalaf, R. Villemaire. Runtime Verification of Web Service Interface Contracts. Computer, 43:59–66, 2010.
26. K. W. Hamlen, G. Morrisett, F. B. Schneider. Certified In-Lined Reference Monitoring on .NET. In 2006 Workshop on Programming languages and Analysis for Security, ACM 2006.
27. F. Han, J. O. Blech, P. Herrmann, and H. Schmidt. Model-based Engineering and Analysis of Space-aware Systems Communicating via IEEE 802.11. To appear in 39th Annual International Computers, Software & Applications Conference (COMPSAC), IEEE Computer, 2015.
28. J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and Matthew Parkinson. Behavioral Interface Specification Languages. ACM Comput. Surv. 44, 3, Article 16, 2012.
29. P. Herrmann. Trust-Based Protection of Software Component Users and Designers. In 1st International Conference on Trust Management, vol. 2692 of LNCS, pages 75–90, Springer-Verlag, 2003.
30. P. Herrmann, J.O. Blech, F. Han, H. Schmidt. A Model-based Toolchain to Verify Spatial Behavior of Cyber-Physical Systems. In 2014 Asia-Pacific Services Computing Conference (APSCC), IEEE Computer.

31. E. B. Johnsen and R. Hähnle and J. Schäfer and R. Schlatte and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In 9th Intl. Symposium on Formal Methods for Components and Objects 2010. Springer-Verlag, 2010.
32. F. A. Kraemer and P. Herrmann. Automated Encapsulation of UML Activities for Incremental Development and Verification. In Model Driven Engineering Languages and Systems (MoDELS), vol. 5795 of LNCS, pages 571–585. Springer-Verlag, 2009.
33. F. A. Kraemer and P. Herrmann. Reactive Semantics for Distributed UML Activities. In Joint WG6.1 International Conference (FMOODS) and WG6.1 International Conference (FORTE), vol. 6117 of LNCS, pages 17–31, Springer-Verlag, 2010.
34. F. A. Kraemer, V. Slåtten and P. Herrmann. Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services. Journal of Systems and Software, 82(12):2068–2080, 2009.
35. E. A. Lee, Y. Xiong. A Behavioral Type System and its Application in Ptolemy II. Formal Aspects of Computing, 16(3):210–237, 2004.
36. B. Meyer. Applying "Design by Contract". Computer, 25(10):40–51, 1992.
37. P. O'Neil Meredith, D. Jin, D. Griffith, F. Chen, G. Roşu. An Overview of the MOP Runtime Verification Framework. International Journal on Software Techniques for Technology Transfer, Springer-Verlag, 2011.
38. C. Prehofer. Behavioral Refinement and Compatibility of Statechart Extensions. In Formal Engineering approaches to Software Components and Architectures. Electronic Notes in Theoretical Computer Science, 2012.
39. F. B. Schneider. Enforceable Security Policies. ACM Transactions on Information and System Security, 3:30–50, 2000.
40. O. Shaer and E. Hornecker. Tangible user interfaces: past, present, and future directions. Foundations and Trends in Human-Computer Interaction 3, no. 12, pp. 1 – 137, 2010.
41. F. Souza, D. Lopes, K. Gama, N. Rosa, R. Lima. Dynamic Event-Based Monitoring in a SOA Environment. In On the Move to Meaningful Internet Systems, vol. 7045 of LNCS, Springer-Verlag, 2011.
42. M. Spichkova, J. O. Blech, P. Herrmann, and H. Schmidt. Modeling Spatial Aspects of Safety-Critical Systems with FOCUSST. In Model-Driven Engineering, Verification, and Validation in MDE, Satellite Event of MoDELS2014, pages 49–58, vol. 1235 in CUR-WS Proceedings, Valencia, 2014.
43. C. Szyperski. Component Software — Beyond Object Oriented Programming. Addison-Wesley Longman, 1997.
44. M. Wenger, J. O. Blech and A. Zoitl. Behavioral Type-based Monitoring for IEC 61499. To appear in Emerging Technologies and Factory Automation (ETFA), IEEE, 2015.