

Preparatory Reflections on Safe Context-adaptive Software (Position Paper)

Dominik Grzelak^a and Uwe Aßmann

Software Technology Group, Technische Universität Dresden, Germany

Keywords: Context-adaptive Software, Formal Models, Software Verification, Future Informatic Systems.


Abstract: Mobile technology and the Internet of Things promise to deepen the interaction between people, services, and physical devices. Digital solutions for these prospective computing systems are not only radically changing the user experience but also the software engineering process. Without a doubt, software complexity enormously increases, and prospective systems become challenging to develop, maintain, and verify. The user's reliance on safety-critical software systems is a serious element in any software engineering process where the absence of bugs must be ensured, and malfunction ruled out. Software that is not safe, i.e., the software's behavior does not comply with a specification, could cause loss of profits or, in the worst-case, harm people. Software safety is an ongoing but mostly academic research field incorporating formal methods to prove the correctness of a program using mathematical methods. In this spirit, we examine the promising context-aware computing and model-driven development paradigms that have directed the development of fog computing and IoT platforms alike. Furthermore, we aggregate viable requirements for computational context models to be employed both for computation and also reasoning about the correctness of applications.

1 INTRODUCTION

When we think about complex systems, we may be inclined to consider only biological, chemical, or physical instances in the first place. Apparently, present-day and prospective *digital anthropomorphic systems*, where we allow us to subsume terms such as mobile computing, Internet of Things (IoT), fog computing, Tactile Internet or ubiquitous systems for the sake of convenience, "will challenge our understanding" (Milner, 2009, p. x) to a much greater extent than before (see also (Baier and Katoen, 2008)). "Context-aware adaptation is an important feature for pervasive computing applications" (Grassi and Sindico, 2007, p. 69), and in this regard, the integration of context introduces further software engineering challenges, which are not present in traditional systems (Henriksen and Indulska, 2006). The reasons are, which will be apparent in the following (see Sec. 1.1 and Sec. 1.2), that software systems inherit a new dimension of complexity, meaning both functional and structural (see (Furrer, 2019)), and as an inevitable consequence, lead to higher development time, hence, costs. For that reason, software engineers continu-

ously trying to shift the boundaries in order to manage complex systems that we do not yet fully understand. A lot of work has been done on investigating methods on how to better cope with the increasing complexity, such as object-oriented programming, modularity, reusability and formal semantics.

In line with this and from a software engineering point of view, we wish to address two crucial qualities of ubiquitous systems in this paper, which are, in our opinion, how to **specify** and **verify the behavior of ubiquitous systems** by means of **formal models**. Under these circumstances, we wish to provide an alternative but complementary approach for application and system development. In this respect, context-aware computing is introduced in Sec. 2, where we explain concrete concepts on how to incorporate context information in applications. Therein, we also present the main features of the *model-driven development* approach in Sec. 2.1, which provides the "glue" for formal models to be practically applied in the software world. In particular, we focus on **computational context models** that allow not only to specify and analyze the interactions of systems with their environment but also to be used for *computation* and *verification* (Sec. 3). Finally, we conclude our paper in Sec. 4.

^a  <https://orcid.org/0000-0001-6334-2356>

The main question that this paper raises is how to improve the overall **code quality** of **context-adaptive software** in ubiquitous systems concerning the *correctness of a program*. We must be committed to communicate this quality criterion properly. Ideally, we have a proof of correctness and other software engineers can understand and verify it. Technically, error checking is much more difficult on large-scale distributed systems than to check a single function in a program. The reliance on the functioning of such systems (cf. (Baier and Katoen, 2008)) is an important point to be considered which goes hand in hand with the objective of bug absence. The presence of errors in software systems is not only annoying. It can, in the most trivial scenarios, e.g., simply freeze the client application, impact the performance, or increase the power consumption of embedded devices, but can also harm people in the worst case.

In the following, we outline the development of IoT, before we wish to describe the implications for software engineering in detail with respect to complex IoT systems. We begin with some technical insights from various analyst's reports (Evans, 2011; Bradley et al., 2013; Winslow et al., 2018).¹

1.1 Current Situation

Dave Evans, a former Cisco expert on the IoT, evaluated the growth rate of IoT and outlined its current state as of 2011, before continuing to provide some rough estimates about the future development and impact of IoT. Therefore, Cisco used research data presented in (Zhang et al., 2008). At the level of autonomous systems (AS), Zhang et al. examined Internet routing data in six-month intervals for the period spanning 2001-2006. In their network model, ASs are connected by links. As two ASs in this model link to each other, they make decisions based not only on the physical connection but also on commercial agreements between two systems. The findings estimated an exponential growth rate of the Internet in the sense that it doubles in size every 5.32 years. Then, Cisco's methodology of applying this constant to the "number of connected devices at a point in time" (Evans, 2011, p. 10) yielded their estimates for the number of devices per person until 2020: Accordingly, based on data available from U.S. Census Bureau (2010) and Forrester Research (2003), the author determined that

¹Note that we are not interested in the value at stake, new economic value chains, revenues for industries, companies, manufacturers, or network operators. However, we want to provide some realistic figures that reflect the growth rate of IoT to make the complex system dilemma evident and what apparent effects this will have for software engineering.

approximately 500 million devices connected to the Internet existed with a world population of 6.3 billion people in 2003. In 2010, both figures grew to 12.5 billion devices and 6.8 billion people, which means that the number of connected devices per person was 1.84 in 2010. Taking only the people connected to the internet into account, resulted in approximately 2 billion people for 2010, meaning, 6.25 devices could be assigned per person (Evans, 2011). By the end of 2020, Evans projected 50 billion connected devices, which makes roughly 6.58 devices per person, assuming a world population of 7.6 billion people.

In the second study (Bradley et al., 2013), other Cisco-related analysts investigated the main driving factors and technology trends for the Internet of Everything (IoE) and, among other topics, the growth rate of the number of devices connected to the Internet. The authors described that in 2000 there were 200 million devices connected to the Internet, which is in accordance with the study above. However, the figures are slightly relativized—the authors stated that in 2013 10 billion connected devices existed from approximately 1.5 trillion devices globally available (Bradley et al., 2013). In their opinion, this number is expected to grow to 50 billion connected devices until the end of 2020, which yields 6.25 devices per person. Further, the authors highlighted the fact that most of the devices are still unconnected resulting in "approximately 200 connectable things per person" (Bradley et al., 2013, p. 2). Their estimation, drawn from Cisco studies in 2013, is that 99.4% of all available IoT devices are still not connected.

The conclusion is consistent with the one of other analysts (Winslow et al., 2018). They also expect a continued "growth [...] into the next decade [...] shaped by [...] the Internet of Things" (Winslow et al., 2018, p. 1). Projections provided by the analysts show that by 2020 roughly 26 IoT devices can be assigned per person from approximately 200 billion connected devices (4 times higher than in (Evans, 2011)), assuming the same world population as above. In particular, the authors conclude that cloud computing concepts alone (which mostly resemble centralized architectures) are no appropriate and feasible computing models when considering "the exponential growth of data created at the edge" (Winslow et al., 2018, p. 6). The authors reported that the amount of data generated at the network edge "will exceed 40 trillion gigabytes by 2025" (Winslow et al., 2018, p. 3) according to projections of IDC white paper (Turner et al., 2014). As mentioned by the authors, they provide strong evidence that a substantial amount of data will be generated and processed in the network edge instead by cloud-

data centers (Winslow et al., 2018) to alleviate the core limitations of purely cloud-centric IoT platforms such as bandwidth, connectivity, latency and context-awareness (see (Bonomi et al., b; Bonomi et al., a; Winslow et al., 2018)).

1.2 Implications on Software Engineering

Current systems have reached or in some way approaching a complexity whose consequences are increasingly difficult to control and understand (Murer et al., 2008; Milner, 2009). Fast software implementations of business requirements are easily at the expense of quality (Murer et al., 2008). Furrer (Furrer, 2019) compiles three general key challenges on the difficulty of the systems engineering process, namely, **change**, **complexity**, and **uncertainty**: (i) The aspect *change* is characterized by changing business requirements, technology updates, or maintenance processes (Furrer, 2019). Moreover, the time cycles to implement necessary changes becoming shorter (Furrer, 2019). Imagine a normal development process, where a team of developers may submit hundreds of changes to upstream software each day. As a result, fast incremental software changes may lead to code redundancy or violation of architecture principles to be followed, which are essential for a uniform code basis and the communication among team members. Each change may introduce new incompatibilities which additionally increases the complexity; (ii) *Complexity* issues may arise from incompatibility with other software, either caused by changing interface specifications over time or heterogeneity of system and software components, to mention a few. Especially when building a specialized software ecosystem, such as in computer-assisted software engineering environments (Wasserman, 1990), tool integration becomes difficult because of many heterogeneous vendor-specific tools. Thus, integration techniques have to be developed, or the usage of non-proprietary formats has to be promoted, for example. Generally, complexity can be classified as structural (architectural) and functional complexity, where the former takes the number of individual system parts and the intensity of their relationships into account, whereas the latter is "measured as the size of the functionality of their parts and interfaces" (Furrer, 2019, p. 24); (iii) *Uncertainty* involves incomplete requirements (e.g., due to the market) that may lead to inadequate decision-making processes, and also includes internal activities during the software's operation (Furrer, 2019). Particularly cyber-physical systems operate in uncertain and unstructured environ-

ments where, however, the system must adequately function, despite the fast-changing and unpredictable surrounding (Furrer, 2019). This also applies to context-aware systems (CAS). As previously mentioned, such a system includes context information to adapt to changing situations autonomously. Thus, they are able to make a smart decision to some extent, which attenuates some problems of uncertainty.

Although these concepts generally apply to several kinds of systems, there is a substantial difference when developing context-adaptive software as opposed to traditional desktop applications that do not primarily include context information. Additional quality measures for context-aware systems were presented in (Henricksen and Indulska, 2006) in the course of evaluating a case study. From another perspective, these can be reversed to challenges that need to be properly treated. Also mentioned is code complexity, and further **maintainability**, **support for evolution** and **reusability**.

2 THE ADOPTION OF CONTEXT INFORMATION IN SOFTWARE

Having arrived here, we are prepared to discuss different implementation approaches for incorporating and manipulating context information in software applications. First, we introduce the necessary model-driven paradigm that is, in some way, a canonical software-related technological foundation for our further elaborations.

By recalling the standard phases of a software development life cycle, support can come in two forms (see (Dey, 2001; Hennessy, 2004)) in order to perform context-adaptive software engineering. The first approach treats the direct development of the solution architecture; that is, we decide and agree on the exact implementation approach. The other type of assistance comes from a higher level, and related to it are model-driven software development practices. Therein, models are primarily used for the specification, implementation, and deployment, mostly supported by automated processes (such as code generation and model transformation). Formal languages or mathematical theories enable this direct model usage (Hennessy, 2004). Both main branches on how to generally adopt context in software are consolidated into a taxonomy and extended with subcategories. The taxonomy is depicted in Fig. 1 and shall be explained in the following.

Beneficial is the fact that both of these approaches are not only compatible but also complementary. Even though general architecture frameworks and dis-

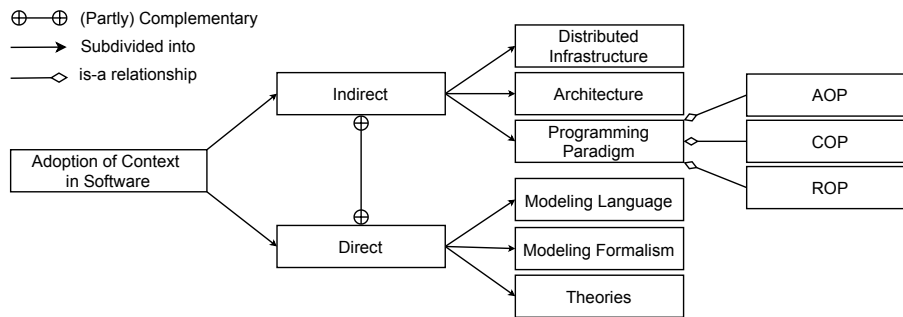


Figure 1: Taxonomy of approaches for context adoption in software (reduced version for clarity). AOP refers to aspect-oriented programming, COP is context-oriented programming and ROP means role-oriented programming. Consolidated and extended after (Dey, 2001; Hennessy, 2004; Broman et al., 2012). See (Broman et al., 2012) for the differences between modeling formalism and language.

tributed infrastructures often limit themselves to the development of particular functionalities or force the commitment of design principles, the context modeling approach does not introduce any arbitrary data structure that is neither in conflict with initial constraints nor requirements.

2.1 Model-driven Engineering

Model-driven engineering (MDE) provides a set of guidelines and instructions to be applied for the software engineering process (Brambilla et al., 2017; Staab et al., 2010; Bézivin, 2005). Nearly two decades ago, the Object Management Group introduced the MDE paradigm to "move from code-centric to model-based practices" (Bézivin, 2005, p. 171). It comprises two main concepts, namely, *models* and *transformations*. Models are the primary artifacts and are expressed in some modeling language, e.g., the Unified Modeling Language (UML). Transformations are a means to reconfigure the model (i.e., provide operations on models). In MDE everything is regarded as a model, even the modeling language is specified by a *meta-model* (refer to the four-layer modelling architecture in (Atkinson and Kuhne, 2003) or see (Bézivin, 2005)).

Models "are no longer mere (passive) documentation" (Ehrig et al., 2006, p. 7) utilities and enable the utilization of sophisticated methods such as syntactical validation and model checking (see (Ehrig et al., 2006; Brambilla et al., 2017) and Sec. 3.2). Models allow to reuse and adapt software (or its components) to different situations (e.g., by transformations), which enables to speed up the development process, eases maintenance and alleviates common errors early in the design phase by using formal models. Moreover, the behavior of a system may be specified by transformation rules. In this regard, higher-order graph transformation rules can be employed to

model self-adaptive systems (Machado et al., 2015).

2.2 Context as a Resource

First, we wish to briefly explain the extremely general term context before we examine the approaches mentioned above. No clear boundary exists what the term context means, hence many loose definitions exist in the scientific literature. However, some key definitions are widely accepted, e.g., (Abowd et al., 1999; Dey, 2001; Abowd et al., 2002), where we wish to adopt the one of Dey (Dey, 2001): "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves" (Dey, 2001, p. 5).

In the last decades, considerable efforts have been undertaken to address this challenge, where context-awareness has been identified as a fundamental requirement for the creation of *robust, intelligent and adaptive applications* (Schmidt et al., 1998; Abowd et al., 1999), and have quickly become a multidisciplinary research field.

2.3 Infrastructures, Architectures and Programming Languages

The main element of this approach is to use ready-made "tools" that assist one in managing the various steps of the software's development lifecycle. Such "tools" can facilitate the development, distribution, and adoption of software for multiple purposes and platforms. Using architectural frameworks also helps teams to better communicate through a common technology. Moreover, they make the maintenance and development process more flexible and efficient when changes are introduced and avoids writing repeated

code. Furthermore, they alleviate common mistakes by providing best practices through their frame of reference. Architectural frameworks and distributed infrastructures manifest themselves in various forms, e.g., graphical and interactive, tool-based, or APIs. They provide software engineers with the necessary frame. We present some examples of recent developments; the list is not exhaustive.

A software toolkit for mobile sensor-based applications is presented in (Grzelak et al., 2019; Grzelak et al., 2020). The toolkit is based on the OSGi standard and modularizes individual components of an application by using the concept of bundles. Further, the toolkit allows adapting an application to the different locality and connectivity of sensor devices by altering the deployment of the application's distributed components.

In (Jaouadi et al., 2018), a model-based approach to develop context-aware systems is proposed that the authors demonstrated by developing a software framework based on a domain-independent context meta-model. The framework provides a Java API "that is capable of capturing the context, observing it in runtime, discovering events that change it and triggering actions to adapt appropriately the running application" (Jaouadi et al., 2018, p. 1170).

In (Kapitsaki and Venieris, 2009), the authors expound the model-driven development of context-aware web services. Their approach allows web services to adapt to context changes. However, often the service's core logic is kept untouched by contexts. Therefore, it is decoupled from related context management components, which enables the reuse of existing functionality and highlighting only on the dependencies that require context information (Kapitsaki and Venieris, 2009). The approach employs UML profiles to model context adaption of a web service.

There is also active research about how context-dependent behavior manifests itself in programming languages. In this regard, we introduce *context-oriented programming* (COP) with the following definition: "Context-oriented programming proposes a language-level technique to enable dynamic adaptations by the activation of contextual situations sensed from the environment. Context activation triggers the dynamic composition of behavioral adaptations with the running system." (Cardozo, 2018, p. 1). It can be thought of as the continuation of procedural and object-oriented languages (Hirschfeld et al., 2008). One of the very first treatments of context-oriented programming are (Gassanenko, 1998; Keays and Rakotonirainy, 2003), providing a generalized notion of context-oriented programming; further (Hirschfeld

et al., 2008), where the authors present required language concept for COP, namely, layers to express different behavioral variations at run-time.

2.4 Context Modeling

Since Mark Weiser's vision of ubiquitous systems, many researchers have studied the foundations of context modeling (e.g. (Henricksen et al., 2002; Roman et al., 2004; Birkedal et al., 2006; Loke, 2016); this list is not exhaustive). This has resulted in a large quantity of different model kinds being developed. Specifically, context models can be divided into 8 different categories: object-role based models, spatial models, ontology-based models, key-value models, object-oriented models, markup scheme models, graphical models, and logic-based models (Bettini et al., 2010; Strang and Linnhoff-Popien, 2004; Bolchini et al., 2007). We may further classify each model as informal or formal. Formal context models are especially useful as they allow reasoning about contexts. The spectrum of categories clearly shows the importance of context models in this research area for future software systems. For the sake of shortness, we cannot give a comprehensive overview of existing context models. Instead, we refer the reader to various surveys and studies conducted so far, e.g., (Henricksen et al., 2002; Strang and Linnhoff-Popien, 2004; Bolchini et al., 2007; Bettini et al., 2010).

These context models are somewhat diametrical to each other and clearly show the wide variety and vibrant landscape. However, "a complete and comprehensive model is still missing" (Chaari et al., 2007, p. 1975). Shortcomings of some of these context models are that they are rather rigid and very specific (e.g., fixed syntax, operations, and semantics), are not defined on a meta-model level, only allow the specification of the context's semantics, or do not scale well because they lack important features such as composition (Grzelak and Alßmann, 2019). This is in accordance with (Henricksen et al., 2002), where the authors observed that most of the architectural frameworks (refer to Sec. 2.3) incorporate informal models that lack the necessary expressive power. Therefore, in the next section, we return to context models that can be expressed algebraically and are more suited to our purpose.

3 MODELING CONTEXT IN SOFTWARE

We focus on a specific aspect of models that can be used not only for the modeling purpose alone but also

for computation and reasoning. This also helps to bridge the gap between models and software development and thus, verification. Therefore, we term these kinds of models **computational context models**. Following (Milner, 2009), a model shall not only be used for the modeling task at hand but also used as a programming language. We define the term as a union of a context model as explained in Sec. 2.4 and a computational model that can be algebraically expressed (e.g., by a process algebra).

To give an impression of this subject, we present some examples. Models that we would like to classify under the term *computational context model* are, e.g., the plato-graphical model in (Birkedal et al., 2006) and the socio-technical model in (Benford et al., 2016). The first paper proposes a context model for the formal modeling of context-aware systems, which comprises three separate models (world, proxy, and application) that can be composed at any time to get the complete view of the system. The second paper presents in detail the modeling of a pervasive outdoor game which comprises four perspectives (computational, physical, human, and technology). The authors demonstrate the analysis of complex interactional phenomena and exploration of possible inconsistencies among the four perspectives of this formal model and developed an application.

3.1 Requirements

We try to determine useful and necessary requirements that a **formal computational context model** must include. In accordance with (Strang and Linnhoff-Popien, 2004; Topcu, 2011; Seshia et al., 2018), we want to promote the following general requirements: (i) **Composability**. Supports to build systems separately and allows a combination later. This feature enables extensible and modularized applications and fosters separation of concerns. (ii) **Validity**. Allows checking whether a model syntactically conforms to a meta-model, thus, ensuring the completeness. Additional constraints can be included for more complex validations. (iii) **Level of Formality**. Means that a model must be precise in terms of the specification of some tasks. On the other hand, it must be easy to use in order to be applied by a user or used as a communication element. (iv) **Verifiability and Reasoning**. Denote that a model must have a formalism that allows performing verification of correctness properties on it and support inferencing of facts by the derivation of other expressions and facts. (v) **Level of Abstraction**. Means that a model must be implementation-agnostic to be maximally interoperable. A high degree of specificity would make the

use of such a model only available for specific applications. (vi) **High Level of Expressiveness**. Allows to create various context semantics, process semantics, describe the distribution and communication of processes at different locations and shall allow the specification of reactive behavior. (vii) **Interoperability**. Allows a model to be incorporated into existing systems or to be used in combination with other models.

3.2 Guaranteeing the Safety of Software at Design-time

Now, we address the quality criterion "safety" as mentioned earlier in the introduction. We present a mechanism on how to ensure the safety of software early in the design phase. Models itself allow, besides model validation and transformations, further model checking and simulation. A lot of work has been conducted in this domain, and we wish to refer the reader to (Seshia et al., 2018; Hoffmann, 2013; Baier and Katoen, 2008) for a more comprehensive overview of this subject of model checking. In the following, we give a brief outline of one particular formal method.

3.2.1 Software Verification

Verification means to ensure that a program at design-time or run-time possesses the desired or required properties according to a specification. Using verification, one is able to detect errors that may not be detected by traditional test and analysis techniques (see (Hoffmann, 2013)) such as unit tests. Formally, in the process of verification, a program (called *implementation*), is checked against a set of constraints (called *specification*) (see (Baier and Katoen, 2008; Hoffmann, 2013)), and we write $I \models S$. These properties are necessary to verify that a system is able to meet the specification.

An advantage of some verification techniques is that they can be utilized in an automated manner. Thus, it found application in industry, in particular, for safety-critical products as a supporting tool (Hoffmann, 2013).

3.2.2 Model Checking

Here, we present a verification technique, called *model checking*, that is commonly employed in the industry. Therefore, model checkers are used for this purpose.

A model checker is a computer program which evaluates state-transition models, such as Kripke

structures, or other similar models (e.g., labeled transition systems, wide reactive systems), against a collection of propositions or constraints, that specify what is required to make the initial assertions of the specification valid.

Well-known model checker tools are, e.g., SPIN (Holzmann, 1997), GROOVE (Rensink, 2004) and PRISM (Kwiatkowska et al., 2011). We observe that model checking is not only a purely academic field but in fact found its application in industry (Hoffmann, 2013). For instance, the model checker SPIN was used for verification of algorithms of the Mars rover Curiosity (Holzmann, 2014). When a simulation is performed using a model checker (assuming a finite state space), it generates an output whether the specification is met or not. Therefore, specialized traversal algorithms are employed (Hoffmann, 2013) that resolve all possible next states from previous ones either as long as some criteria are not violated or by some other constraints (e.g., number of states, transitions, or time limits). Thus, after every iteration, the state space is extended, and the system evolves. If the program contains errors, in the course of the simulation, counterexamples can be generated to provide assistance for the designer, for instance.

According to (Owicki and Lamport, 1982), in the case of concurrent programs, one can verify two key properties: *safety properties* and *liveness properties*. The first verifies that a program never enters an undesired state in the sense that a program becomes non-operational, e.g., because of deadlocks. The second denotes a desirable state of a program that is eventually going to happen, e.g., an exclusive resource can be used by all available processes, or that a program does not terminate unexpectedly.

Liveness properties imply the notion of time, a concept of fairness and deadlock freedom is important for concurrent programs, which includes distributed systems and real-time systems. For instance, real-time systems rely on coordinated operations among their interacting components. Thus, timely coordination is a key element for the correct functioning of those systems (Tripakis and Courcoubetis, 1996). In this regard, one can describe the temporal dynamics of a program by expressions of linear-time properties in the specification. Linear-time properties represent valid traces of a transition system that specifies the desired behavior of a system over time (Baier and Katoen, 2008). Such temporal propositions are often specified using first-order logics such as the linear temporal logic or computation tree logic. We shall not go into more detail here for the sake of space limitations.

Problems. Verification operates on the model-level, which implies that a program must be translated into some kind of a formal model first. This is, at the same time, one of its limits (Hoffmann, 2013). In this process, errors can be introduced, which cannot be detected by the actual program verification. In this regard, we advocate the use of formal modeling approaches early in the design phase, being able to employ them further for model-driven techniques. Another problem is the so-called *state space explosion* problem, where researchers investigated various approaches to reduce this problem, e.g., program slicing (Léchenet et al., 2016), symbolic model checking or the early detection of infinite state traces.

4 CONCLUSIONS

The IoT may offer much in the way of innovation and will further emerge a vast number of new services, but will continue to introduce new difficulties from a software engineering perspective. When the complexity of software systems reaches a critical level, due to the connectedness of our environment, it will become increasingly important for software engineers to have an efficient and effective computing model to use in representing different abstractions of the system to implement, deploy and provision applications. The appliance of model-driven techniques in combination with formal context models may help to cope with the complexity of ubiquitous systems, which directly implies a reduction of the software's complexity, and thus in return, the development time and costs.

Based on the experiences we gained from several research projects (e.g., IoSense², and CeTI³), we may conclude that designing context-aware and adaptive software employing model-driven techniques is a valid and reasonable approach. Context-adaptive software surely presupposes a context system in one way or another. One way to achieve this, is to consider context as part of a platform-independent component in any application, system, or network.

We argue that it is often easiest to explain the intent of any piece of software using the context of the environment in which it is deployed. This approach is applicable for application, system, and infrastructure design. In our view, one unfortunate aspect is that context models have not yet been satisfactorily abstracted to the point that such a software model can be integrated easily into other systems. An important decision on the nature of computational models must

²<http://www.iosense.eu/>

³<https://www.ceti.one/>

be that of the formalism in which they are described. If the precise terminology is too cumbersome or less expressive, it is, on the other hand, more advisable to use a context-oriented programming language or an architectural framework. However, we strongly advocate using models at a reasonable level of abstraction to be able to deal with all unknown interactions in a realistic timeframe in case of internal and external changes. Hence, the choice of a formal model cannot be considered in isolation, and decisions are bounded to trade-offs in each case, which needs to be carefully weighed (cf. (Cafezeiro et al., 2008)).

To conclude our digression, we wish to suggest hierarchical graph models by following (Milner, 2009; Bruni et al., 2014), which are also commonly applied for modeling the software system's structure. Recent developments in the domain of process algebra show that hierarchical models can express the two relevant dimensions *locality* and *communication*, which are two essential elements in ubiquitous systems (Bruni et al., 2014).

ACKNOWLEDGEMENTS

Funded by the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) as part of Germany's Excellence Strategy – EXC 2050/1 – Project ID 390696704 – Cluster of Excellence "Centre for Tactile Internet with Human-in-the-Loop" (CeTI) of Technische Universität Dresden.

REFERENCES

- Abowd, G. D., Dey, A. K., Brown, P. J., Davies, N., Smith, M., and Stegless, P. (1999). Towards a Better Understanding of Context and Context-Awareness. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, HUC '99, pages 304–307. Springer-Verlag.
- Abowd, G. D., Ebling, M., Hung, G., Lei, H., and Gellersen, H. (2002). Context-aware computing [Guest Editors' Intro.]. *IEEE Pervasive Computing*, 1(3):22–23.
- Atkinson, C. and Kuhne, T. (2003). Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41.
- Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. The MIT Press.
- Benford, S., Calder, M., Rodden, T., and Sevegnani, M. (2016). On Lions, Impala, and Bigraphs: Modelling Interactions in Physical/Virtual Spaces. 23(2):9:1–9:56.
- Bettini, C., Brdiczka, O., Henriksen, K., Indulska, J., Nicklas, D., Ranganathan, A., and Riboni, D. (2010). A Survey of Context Modelling and Reasoning Techniques. *Pervasive Mob. Comput.*, 6(2):161–180.
- Birkedal, L., Debois, S., Elsborg, E., Hildebrandt, T., and Niss, H. (2006). Bigraphical Models of Context-Aware Systems. In *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 187–201. Springer, Berlin, Heidelberg.
- Bolchini, C., Curino, C. A., Quintarelli, E., Schreiber, F. A., and Tanca, L. (2007). A Data-oriented Survey of Context Models. *SIGMOD Rec.*, 36(4):19–26.
- Bonomi, F., Milito, R., Natarajan, P., and Zhu, J. Fog Computing: A Platform for Internet of Things and Analytics. In Bessis, N. and Dobre, C., editors, *Big Data and Internet of Things: A Roadmap for Smart Environments*, Studies in Computational Intelligence, pages 169–186. Springer International Publishing.
- Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16. ACM.
- Bradley, J., Barbier, J., and Handler, D. (2013). More Relevant, Valuable Connections Will Improve Innovation, Productivity, Efficiency & Customer Experience.
- Brambilla, M., Cabot, J., and Wimmer, M. (2017). *Model-Driven Software Engineering in Practice: Second Edition*. Morgan & Claypool Publishers, 2nd edition.
- Broman, D., Lee, E. A., Tripakis, S., and Törngren, M. (2012). Viewpoints, Formalisms, Languages, and Tools for Cyber-physical Systems. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, MPM '12, pages 49–54. ACM.
- Bruni, R., Montanari, U., Plotkin, G., and Terreni, D. (2014). On Hierarchical Graphs: Reconciling Bigraphs, Gs-monoidal Theories and Gs-graphs. *Fundamenta Informaticae*, 134:287–317.
- Bézivin, J. (2005). On the unification power of models. *Software & Systems Modeling*, 4(2):171–188.
- Cafezeiro, I., Viterbo, J., Rademaker, A., Haeusler, E. H., and Endler, M. (2008). A Formal Framework for Modeling Context-Aware Behavior in Ubiquitous Computing. In Margaria, T. and Steffen, B., editors, *Leveraging Applications of Formal Methods, Verification and Validation*, Communications in Computer and Information Science, pages 519–533. Springer Berlin Heidelberg.
- Cardozo, N. (2018). A Declarative Language for Context Activation. In *Proceedings of the 10th International Workshop on Context-Oriented Programming: Advanced Modularity for Run-Time Composition*, COP '18, pages 1–7. Association for Computing Machinery.
- Chaari, T., Ejigu, D., Laforest, F., and Scuturici, V.-M. (2007). A Comprehensive Approach to Model and Use Context for Adapting Applications in Pervasive Environments. *J. Syst. Softw.*, 80(12):1973–1992.
- Dey, A. K. (2001). Understanding and Using Context. *Personal Ubiquitous Comput.*, 5(1):4–7.

- Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G. (2006). *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag.
- Evans, D. (2011). How the Next Evolution of the Internet Is Changing Everything.
- Furrer, F. J. (2019). Three Devils of Systems Engineering. In Furrer, F. J., editor, *Future-Proof Software-Systems: A Sustainable Evolution Strategy*, pages 21–44. Springer Fachmedien.
- Gassanenko, M. L. (1998). Context-Oriented Programming. In *euroForth'98*, page 10.
- Grassi, V. and Sindico, A. (2007). Towards model driven design of service-based context-aware applications. In *International Workshop on Engineering of Software Services for Pervasive Environments: In Conjunction with the 6th ESEC/FSE Joint Meeting*, ESSPE '07, pages 69–74. Association for Computing Machinery.
- Grzelak, D. and Aßmann, U. (2019). Bigraphical meta-modeling of fog computing-based systems. In *Proceedings of the International Conference on Discrete Models of Complex Systems (SOLSTICE)*.
- Grzelak, D., Mai, C., and Aßmann, U. (2019). Towards a Software Architecture for Near Real-time Applications of IoT. In *Proceedings of the 4th International Conference on Internet of Things, Big Data and Security - Volume 1: IoTBDS*, pages 197–206. INSTICC, SciTePress.
- Grzelak, D., Mai, C., Schöne, R., Falkenberg, J., and Aßmann, U. (2020). A Software Toolkit for Complex Sensor Systems in Fog Environments. In van Driel, W. D., Pyper, O., and Schumann, C., editors, *Sensor Systems Simulations: From Concept to Solution*, pages 253–282. Springer International Publishing.
- Hennessy, M. (2004). Context-awareness: Models and analysis.
- Henricksen, K. and Indulska, J. (2006). Developing Context-aware Pervasive Computing Applications: Models and Approach. *Pervasive Mob. Comput.*, 2(1):37–64.
- Henricksen, K., Indulska, J., and Rakotonirainy, A. (2002). Modeling Context Information in Pervasive Computing Systems. In Mattern, F. and Naghshineh, M., editors, *Pervasive Computing*, Lecture Notes in Computer Science, pages 167–180. Springer Berlin Heidelberg.
- Hirschfeld, R., Costanza, P., and Nierstrasz, O. (2008). Context-Oriented Programming. *Journal of Object Technology*, ETH Zurich, 7(3):125–151.
- Hoffmann, D. W. (2013). Software-verifikation. In Hoffmann, D. W., editor, *Software-Qualität*, eXamen.press, pages 333–369. Springer.
- Holzmann, G. (1997). The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295.
- Holzmann, G. J. (2014). Mars code. *Communications of the ACM*, 57(2):64–73.
- Jaouadi, I., Ben Djemaa, R., and Ben-Abdallah, H. (2018). A model-driven development approach for context-aware systems. *Software & Systems Modeling*, 17(4):1169–1195.
- Kapitsaki, G. M. and Venieris, I. S. (2009). Model-Driven Development of Context-Aware Web Applications Based on a Web Service Context Management Architecture. In Chaudron, M. R. V., editor, *Models in Software Engineering*, Lecture Notes in Computer Science, pages 343–355. Springer.
- Keays, R. and Rakotonirainy, A. (2003). Context-oriented Programming. In *Proceedings of the 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access*, MobiDe '03, pages 9–16. ACM.
- Kwiatkowska, M., Norman, G., and Parker, D. (2011). PRISM 4.0: Verification of Probabilistic Real-Time Systems. In Gopalakrishnan, G. and Qadeer, S., editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 585–591. Springer.
- Léchenet, J.-C., Kosmatov, N., and Le Gall, P. (2016). Cut Branches Before Looking for Bugs: Sound Verification on Relaxed Slices. In Stevens, P. and Wařowski, A., editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 179–196. Springer.
- Loke, S. W. (2016). Representing and reasoning with the internet of things: A modular rule-based model for ensembles of context-aware smart things. *EAI endorsed transactions on context-aware systems and applications*, 3(8):1–17.
- Machado, R., Ribeiro, L., and Heckel, R. (2015). Rule-based transformation of graph rewriting rules: Towards higher-order graph grammars. *Theoretical Computer Science*, 594:1–23.
- Milner, R. (2009). *The Space and Motion of Communicating Agents*. Cambridge University Press, 1st edition.
- Murer, S., Worms, C., and Furrer, F. J. (2008). Managed evolution. *Informatik-Spektrum*, 31(6):537–547.
- Owicki, S. and Lamport, L. (1982). Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495.
- Rensink, A. (2004). The GROOVE Simulator: A Tool for State Space Generation. In Pfaltz, J. L., Nagl, M., and Böhlen, B., editors, *Applications of Graph Transformations with Industrial Relevance*, Lecture Notes in Computer Science, pages 479–485. Springer.
- Roman, G.-C., Julien, C., and Payton, J. (2004). A Formal Treatment of Context-Awareness. In Wermelinger, M. and Margaria-Steffen, T., editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 12–36. Springer Berlin Heidelberg.
- Schmidt, A., Beigl, M., and Gellersen, H.-w. (1998). There is more to Context than Location. *Computers and Graphics*, 23:893–901.
- Seshia, S. A., Sharygina, N., and Tripakis, S. (2018). Modeling for Verification. In Clarke, E. M., Henzinger, T. A., Veith, H., and Bloem, R., editors, *Handbook of Model Checking*, pages 75–105. Springer International Publishing.
- Staab, S., Walter, T., Gröner, G., and Parreiras, F. S. (2010). Model Driven Engineering with Ontology Technolo-

- gies. In Abmann, U., Bartho, A., and Wende, C., editors, *Reasoning Web. Semantic Technologies for Software Engineering: 6th International Summer School 2010, Dresden, Germany, August 30 - September 3, 2010. Tutorial Lectures*, Lecture Notes in Computer Science, pages 62–98. Springer Berlin Heidelberg.
- Strang, T. and Linnhoff-Popien, C. (2004). A Context Modeling Survey. page 8. UbiComp 2004 workshop on Advanced Context Modelling, Reasoning and Management.
- Topcu, F. (2011). Context Modeling and Reasoning Techniques.
- Tripakis, S. and Courcoubetis, C. (1996). Extending promela and spin for real time. In Margaria, T. and Steffen, B., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 329–348. Springer.
- Turner, V., Gantz, J. F., Reinsel, D., and Minton, S. (2014). The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things.
- Wasserman, A. I. (1990). Tool integration in software engineering environments. In Long, F., editor, *Software Engineering Environments*, Lecture Notes in Computer Science, pages 137–149. Springer Berlin Heidelberg.
- Winslow, P., Fritzsche, J. M., Stabler, P., Rakers, A., Luechow, E., and Hilliker, R. (2018). The Edge Of Glory (or: The Fog Rolls In).
- Zhang, G.-Q., Zhang, G.-Q., Yang, Q.-F., Cheng, S.-Q., and Zhou, T. (2008). Evolution of the Internet and its cores. *New Journal of Physics*, 10(12):123027.

