Charles University in Prague

Faculty of Mathematics and Physics

DOCTORAL THESIS

Jaroslav Keznikl

Dynamic Software Architectures for Resilient Distributed Systems

Department of Distributed and Dependable Systems

Advisor:	Doc. RNDr. Tomáš Bureš, Ph.D.
Study program:	Computer Science
Specialization:	Software Systems

Prague 2014

Acknowledgments

I would like to thank all who supported me during my doctoral studies and without whom this work would not have been possible. First of all, I sincerely thank my advisor Tomáš Bureš for his indispensable guidance, support, and co-authorship of all the included papers. Moreover, I thank him for instilling in me his passion for research. I am also deeply grateful to František Plášil for his unending support in research and his patience, guidance, and collaboration in writing. I thank all the rest of my co-authors for their valuable contributions and all my colleagues at the Department of Distributed and Dependable Systems for creating such an inspiring and friendly working environment. A particular thank-you goes to: Michal Kit, Ilias Gerostathopoulos, and Rima Al Ali for their enthusiasm and fellowship in research; Michal Malohlava, Pavel Parízek, and Viliam Šimko for their valuable advice and encouragement; Lubomír Bulej, Petr Hnětynka, Vojtěch Horký, Pavel Jančík, Pavel Ježek, Jan Kofroň, Andrej Podzimek, Tomáš Poch, Tomáš Pop, Ondřej Šerý, and Petr Tůma for their feedback and fruitful discussions.

Furthermore, I am grateful to Nicklas Hoch and Henry Bensler, who allowed me to spend four exciting months in their team at Volkswagen Research.

My thanks go also to the institutions that provided financial support for my research work. Throughout my doctoral studies, my work was partially supported by the EU project ASCENS 257414 under the ICT priority of the 7th Research Framework Programme, the Grant Agency of the Czech Republic projects P103/11/1489, P202/11/0312, and P202/10/J042, the Charles University institutional funding SVV-2011-263312, SVV-2012-265312, and SVV-2013-267312, the Czech Science Foundation grant 201/09/H057, and the Ministry of Education of the Czech Republic grant MSM0021620838.

Above all, I am deeply in debt to my girlfriend Lucie and my family for their endless patience and encouragement, without which I could not have completed this work.

Annotation

Title	*Dynamic Software Architectures for Resilient Distributed Systems*
Author	Jaroslav Keznikl keznikl@d3s.mff.cuni.cz (+420) 221 914 236
Department	Department of Distributed and Dependable Systems Faculty of Mathematics and Physics Charles University in Prague
Advisor	Doc. RNDr. Tomáš Bureš, Ph.D. bures@d3s.mff.cuni.cz (+420) 221 914 236
Mailing address	Department of Distributed and Dependable Systems Charles University in Prague Malostranské náměstí 25 118 00 Prague, Czech Republic
WWW	http://d3s.mff.cuni.cz/

Abstract

Resilient Distributed Systems (RDS) are large-scale distributed systems that remain dependable despite their very dynamic, open-ended, and inherently unpredictable environments. This combination of system and environment properties makes development of software architectures for RDS using contemporary architecture models and abstractions very challenging.

Therefore, the thesis proposes: (1) new architecture abstractions that are tailored for building dynamic software architectures for RDS, (2) design models and processes that endorse these abstractions at design time, and (3) means for efficient implementation, execution, and analysis of architectures based on these abstractions.

Specifically, the thesis delivers (1) by introducing the DEECo component model, based on the concept of component ensembles. Contributing to (2), the thesis presents the Invariant Refinement Method, governing dependable, formally-grounded design of DEECo-based architectures, and the ARCAS method, focusing on dependable realization of open-ended dynamic component bindings typical for DEECo. Furthermore, it pursues (3) by presenting a formal operational semantics of DEECo and its mapping to Java in terms of an execution environment prototype – jDEECo. Additionally, the semantics is used as a basis for formal analysis via model checking. Finally, the thesis validates DEECo by presenting a dynamic architecture of an RDS ensuring adaptive task deployment in ad-hoc cloud systems.

Keywords

Software architecture, Adaptation, Component model, Formal semantics

Anotace

Název práce	*Dynamické Softwarové Architektury pro Resilientní Distribuované Systémy*
Autor	Jaroslav Keznikl keznikl@d3s.mff.cuni.cz (+420) 221 914 236
Katedra	Katedra distribuovaných a spolehlivých systémů Matematicko-fyzikální fakulta Univerzita Karlova v Praze
Školitel	Doc. RNDr. Tomáš Bureš, Ph.D. bures@d3s.mff.cuni.cz (+420) 221 914 236
Adresa	Katedra distribuovaných a spolehlivých systémů Univerzita Karlova v Praze Malostranské náměstí 25 118 00 Praha
WWW	http://d3s.mff.cuni.cz/

Abstrakt

Resilientní Distribuované Systémy (RDS) jsou ty rozsáhlé distribuované systémy, které fungují spolehlivě navzdory svému značně dynamickému, otevřenému a z principu nepředvídatelnému prostředí. Takováto kombinace systémových vlastností a vlastností prostředí však velmi ztěžuje vývoj softwarových architektur pomocí dnes dostupných softwarových modelů a abstrakcí.

Proto se tato dizertační práce snaží přinést: (1) nové abstrakce, které jsou speciálně uzpůsobeny potřebám dynamických softwarových architektur RDS, (2) softwarové modely a procesy, které usnadňují použití těchto abstrakcí během vývoje, a (3) prostředky pro efektivní implementaci, provoz i analýzu softwarových architektur postavených na těchto abstrakcích.

Tato práce řeší bod (1) zavedením komponentového modelu DEECo, jenž je založen na konceptu ensemblů komponent. Práce dále přispívá k (2) představením metody Invariant Refinement Method, která zajišťuje spolehlivý formalizovaný vývoj softwarových architektur postavených na DEECo, a uvedením metody ARCAS, která se soustředí na spolehlivou realizaci dynamických spojení komponent, typických pro DEECo. Práce se věnuje (3) prostřednictvím formalizace operační sémantiky komponentového modelu DEECo a projekcí této sémantiky do jazyka Java formou prototypu běhového prostředí – jDEECo. Tato sémantika je dále využita jako základ pro formální analýzu pomocí model checkingu. Nakonec práce validuje DEECo na příkladu dynamické softwarové architektury pro RDS, který zajišťuje adaptivní deployment v ad-hoc cloud systémech.

Klíčová slova

Softwarová architektura, Adaptace, Komponentový model, Formální sémantika

Contents

Chapter 1

Introduction

1.1 Towards Resilient Distributed Systems

The recent significant increase in the ubiquity and connectivity of smart embedded and mobile computing devices has opened new possibilities for addressing social and environmental challenges, such as ambient assisted living, smart city infrastructures, emergency coordination, and environmental monitoring. In particular, this progress has provided the infrastructure necessary for building pervasive software systems that harness diverse data sources to respond to and influence the real world. An example could be an intelligent traffic control that gathers data from cars and other sensors in a city so as to navigate the cars, control traffic lights, and manage parking allocation.

As such, in addition to being closely bound to the real world, these systems typically need to be decentralized, distributed, and heterogeneous. Moreover, they have to cope with their very dynamic, open-ended, and inherently unpredictable environments (e.g., road networks, emergency sites). In the context of this thesis, we will call the systems featuring these properties *Resilient Distributed Systems* (RDS).

An important desired feature of RDS is that they are distributed on a large scale, while relying on a relative autonomy of individual system components, are inherently dynamic so as to adapt to recurrently changing situations in their environment, and, most importantly, are highly dependent on software, i.e., they are software-intensive systems [BB12, HRW08]. This means that software is by far their most important and the most complex constituent.

Developing large-scale RDS via systematic software engineering approaches is a notoriously difficult task. This stems from the fact that the requirements of RDS invalidate certain critical assumptions that typically hold in software engineering of conventional distributed systems. A distinct challenge of RDS is that their software undergoes continuous modifications as the real world evolves: components appear and disappear as devices enter/exit the system, components form and dissolve cooperation groups as they start/finish a particular joint activity, and communication links are established/released depending on the actual network connectivity. Consequently, communication between the components is opportunistic and there are no guarantees regarding the stability and reliability of the established communication links. The network topology itself is extremely dynamic and often relies on ad-hoc means without any managing infrastructure.

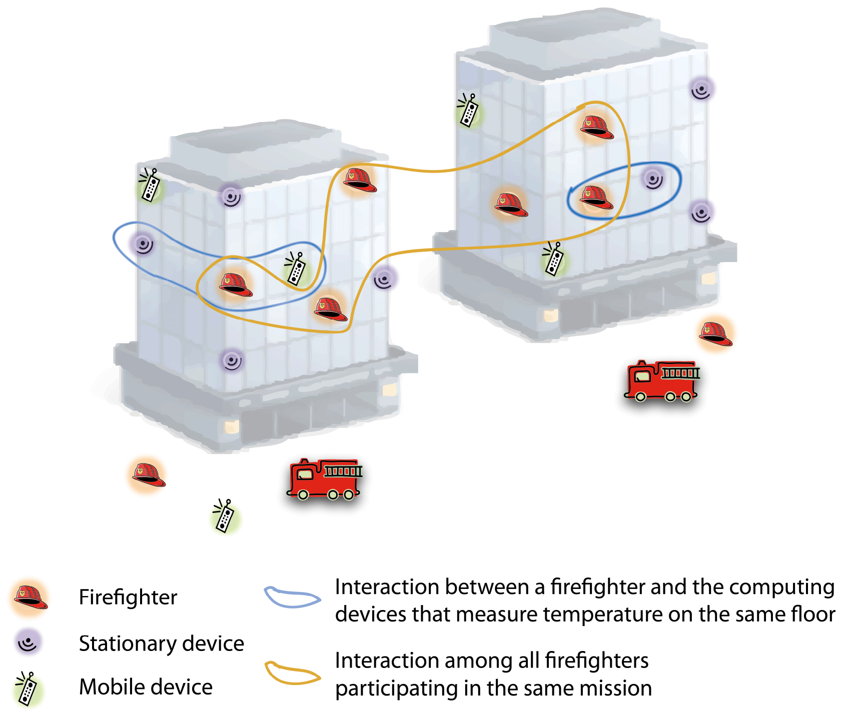

Firefighter
Stationary device
Mobile device
Interaction between a firefighter and the computing devices that measure temperature on the same floor
Interaction among all firefighters participating in the same mission

Figure 1. RDS example: firefighter coordination scenario. Adopted from [BGAA14].

Moreover, the scale, dynamicity, and complexity of RDS introduces emergent behavior (i.e., behavior that comes about as the joint product of behaviors and interactions of many elements of a system). Finally, a particularly important concern in RDS is dependability, as the close connection to the real world frequently renders the functionality of RDS safety-critical.

At this point, it is important to note that, from a wider perspective, distributed software systems closely interacting with their physical environment are called Cyber-Physical Systems (CPS) [RLSS10]. To this end, the above-outlined area of RDS partially overlaps and shares many common software-engineering challenges with CPS. Nevertheless, whereas the key concerns of RDS are distribution, dynamism/mobility, and safety-criticality while acknowledging the role of the physical environment, the key concern of CPS is the actual control of the physical environment. For the purpose of this thesis, we will consider CPS as a particular case of RDS.

1.1.1 Example of a Resilient Distributed System

To better illustrate the context and challenges of RDS, the following text describes a scenario (Figure 1) that is based on the firefighter emergency response operations [YYP13]. In the scenario, firefighters belong to teams corresponding to the mission at hand. A critical requirement is an efficient dissemination of information among team members – every member has to be notified about important events and threats (e.g., low oxygen level in a particular room, firefighter in danger because of high temperature level). To achieve this, the firefighters are equipped with mobile computing devices that are integrated into their personal protection equipment and communicate via wireless network interfaces. Using these devices, the firefighters coordinate within and across several mission sites (e.g., buildings) while also taking advantage of the heterogeneous stationary and mobile computing devices existing in their vicinity (Figure 1). For instance, in addition to its team members, each firefighter interacts with all the reachable devices that provide a temperature-monitoring feature and are located on the same floor. A key challenge of this scenario is that the whole firefighter coordination system needs to operate in a distributed and decentralized manner, as there is no centrally managed infrastructure on the deployment site. In addition, the communication links continuously change as the firefighters move across the floors, mobile devices get out of reach, new firefighters are assigned to teams, etc. Finally, the system needs to be dependable and predictable, as the firefighters' safety depends on it.

1.2 Software Architectures and RDS

The distinct properties of RDS bring about a number of challenges pertaining to all phases of software design and development. Although some of the challenges are researched in various other domains, such as the domain of middleware for mobile ad-hoc networks [FGR+07], this thesis focuses specifically on the area of *software architectures*, since we believe that it provides "the required level of abstraction and generality to deal with the challenges posed" [KM07]. Consequently, given the fact that the challenges of RDS are mainly related to dynamic and open environments, the particular focus of this thesis is on *dynamic* software architectures.

As such, software architecture is the set of principal design decisions made about a system [TMD10]. Specifically, it governs (i) the high-level software structure, (ii) interaction of system components, (iii) core decisions about functional behavior, (iv) non-functional properties, and (v) basic aspects of the system's implementation.

Naturally, there is a large number of approaches to software architectures, including various software-architecture models and software-engineering methods, many of which partially address the above-outlined challenges. However, as we will show, none of them can be readily applied for building RDS. In the remainder of this section, we quickly look at how some of these approaches address particular architecture-related challenges of RDS. As an aside, a more in-depth description of the hereby-discussed approaches is presented in Chapter 2.

One of the software-architecture approaches relevant to RDS is *component-based development* (CBD). CBD originates from the necessity of reuse and relies on separation of concerns to tame the complexity of building and maintaining large applications [CL02, HC01, Szy02]. Despite being successful in many domains, ranging from enterprise applications to embedded systems, CBD has a number of limitations when applied in the domain of RDS. First, CBD typically makes the assumption of centralized component deployment. Second, there is no widely accepted approach regarding dynamic changes in a component-based architecture. Third, CBD assumes reliable communication among components, which is not plausible for many RDS. Lastly, there is a strong conceptual reliance of the components in a system on each other, which hinders the resilience of the system as a whole (i.e., the system might stop working when one component becomes unavailable). The above holds for both academic component systems, such as Fractal [BCL+06] and SOFA 2 [BHP06], and industrial ones, such as EJB [19] and CCM [WSO01][18].

In an attempt to deal with the issues of centralized ownership and deployment of component-based systems, as well as their static architectures, a shift has been observed from components to services [HS05]. *Service-oriented computing* (SOC) provides the means for dynamic service binding in face of changing service availability that are typically based on a well-known service discovery mechanism. It also clearly separates the responsibilities of the two sides involved in a distributed interaction – the service provider and the service consumer (end-user or another service), thus rendering the service-based architectures loosely coupled. Service platforms such as iPOJO [EHL07] and service coordination languages such as WS-BPEL [JEA+07] feature the above advantages. Nevertheless, there are still serious limitations in the direct employment of SOC in the domain of RDS. Specifically, there is a strong reliance on a (typically) centralized service platform for service discovery and binding. Also, SOC still assumes already-connected services to communicate seamlessly. This makes service-based systems yet not suitable for RDS.

To mitigate the strong reliance of components/services on each other, *agent-oriented computing* (AOC) has introduced an abstraction that brings conceptual autonomy to loosely coupled system components – *software agents*. Consequently, multi-agent systems [SLB09] have brought the autonomy to architecture organization and allowed building self-organized systems that are heavily dynamic. To this end, AOC provides useful concepts (e.g., groups and roles [FGM04]) and models (e.g., Belief-Desire-Intention architectural model [RG+95]) for designing complex autonomic systems, such as RDS. However, the problem yet to be tackled is that software agent implementations and the related agent platforms (e.g., JACK [BRHL99] or JADE [BPR01]) do not translate the conceptual autonomy of the agent notions into proper software engineering constructs that deal with real-life constraints of autonomous behavior. In particular, these platforms and the related development methods [BPG+04] rely on the assumption of explicit (message-oriented) communication, existence of a (centralized) platform, and relatively stable communication links among the agents, which is not plausible for RDS. Moreover, AOC primarily focuses on harnessing autonomic agent behavior, rather than on software architecture design.

This is partially addressed by the concept of *service-component ensembles* [HRW08], which has been the key notion of the FP7 project ASCENS [5]. The underlying ideas are embodied by the agent-oriented coordination language SCEL [DNFLP13, DNLPT14], where attribute-based communication is employed to model dynamic interaction of service components. However, although providing powerful coordination concepts with precise semantics, this body of work still falls short in providing dedicated support for appropriate software architecture abstractions, their implementation, and the related software-engineering processes.

Addressing the challenges of RDS from the perspective of architecture (self-) adaptation and exploiting the principles of *control engineering* [MPS08, PCHW12], a significant body of work has been focusing on instantiation of a generic *feedback-loop* scheme (e.g., Monitor-Analyze-Plan-Execute – MAPE-K [KC03]) at the level of software architecture [BSG+09, CDLG+09, DLGM+13]. Examples are layered self-managing architectures based on explicit adaptation-goal management [KM07, TGEM10] and approaches that apply architectural-variability models at runtime [MBNJ09]. Nevertheless, these approaches primarily target the high-level architecture self-adaptation aspects (i.e., architecture monitoring, adaptation planning, and adaptation execution), rather than the autonomic, dynamic, and distributed operation of RDS in the physical environment.

1.3 Problem Statement

Looking at the contemporary approaches to software architectures and the related software engineering methods outlined in Section 1.2, the specifics of RDS make it difficult to employ these approaches directly. In fact, the task of engineering dynamic software architectures for RDS reaches the threshold when it is disputable whether we are still dealing with a variant of traditional software engineering or whether we are encountering a new paradigm in computing. This is also reflected by the fact that software engineering of systems akin to RDS is a widely recognized problem domain targeted by ongoing research agendas (e.g., EU Research and Innovation program Horizon 2020).

The main shortcoming of the existing approaches is that they have been designed to work under assumptions that are simply unrealistic in the domain of RDS. As discussed from the perspective of CPS in [GKB+14], the violated assumptions include, but are not limited to, the assumption of static physical structure, stable connections, independence on physical location, clique connectivity, focus on reactive behavior, stateful communication, and controlled architecture dynamism. This calls for tailored software architecture models, their implementations, and software-engineering methods that address and potentially take advantage of the RDS specifics.

To this end, the key concern of this thesis is to tackle this issue by addressing specifically the following software-architecture challenges:

C1 **Recurrent context-driven adaptation**. In the context of this thesis, the most critical of the challenges pertains to the inherent dynamism and unpredictability of the physical environment of RDS. In fact, the physical substratum is continuously evolving as mobile computing devices move in the environment. This means that software architecture has to adapt automatically and dynamically to the recurrently changing situations in the physical environment. In addition, components are no longer purely virtual software entities but often have their representations in the physical environment and thus have to be treated as autonomous entities that cannot always be explicitly managed (e.g., created or connected). Consequently, the task of architecture adaptation can be no longer perceived as a single step or a temporary phase that is fully controlled by the decisions of the system itself or its administrator. Instead, it has to be a continuous activity that is driven by the observable changes in the physical, as well as virtual, environment. For this purpose, the architecture also needs to allow for continuous interaction with the physical environment.

C2 **Resilience to unstable communication links.** The mobility of the computing devices in the physical environment of RDS renders communication links inherently unstable. A communication link may become (un)available at any time. Hence, errors in communication are the rule, not the exception, and thus they cannot be handled as such any more. The property of unstable communication links has to be acknowledged and ideally reflected in software architecture. From the point of view of an individual system component, the architecture needs to allow for correct functioning of the component even when fully detached from the rest of the system, i.e., in full autonomy. From the global perspective, the architecture needs to be decentralized.

C3 **Scalability and open-endedness.** Since heterogeneous components may join/leave an RDS (or its partition) as a result of the mobility of the computing devices they are deployed on, the set of components that may constitute the RDS is open and essentially unbounded. Consequently, software architecture of RDS has to be open-ended so as to support new components coming from devices appearing in the physical environment. This also implies that the architecture needs to be scalable at both design time and runtime. The actual scale of the architecture is limited only by the distribution of the computing devices in the physical environment.

C4 **Dependability.** Being frequently safety-critical, RDS require software architecture to provide a certain degree of dependability – both at the design time and runtime. This implies the need for well-defined formal semantics of architecture models and their implementations, as well as formally grounded analysis methods exploiting these semantics. This requirement itself is not unique; however, a combination of dependability with open-endedness, autonomic behavior, distribution, and self-adaptivity is very challenging, since these concerns are to an extent contradictory. While open-endedness and self-adaptivity typically require facilitating emergent behavior, addressing dependability needs describing and limiting the emergent behavior.

In order to manage the scale and dynamism of RDS, these challenges need to be addressed at both design time, in terms of appropriate architecture models and software-engineering methods, and runtime, in terms of appropriate computational models and execution environments.

1.4 Research Goals

Responding to the challenges presented in Section 1.3, this thesis focuses on the area of dynamic, self-adaptive software architectures for RDS. Specifically, the thesis aims at providing appropriate software architecture models and abstractions that would address these challenges. The primary intention is to adopt the ideas of component-based development [CL02, HC01, Szy02], agent-based computing [SLB09], and ensemble-based systems [HRW08], while focusing on the software engineering aspects.

Since this area is very broad, the thesis primarily focuses on clarifying the crucial aspects concerning architecture abstractions, their semantics, and the related software-engineering concerns. The thesis thus targets the following research goals:

G1 The first goal is to propose **architecture abstractions** that are tailored for designing dynamic architectures for RDS. The focus is on a proper semantics of these abstractions w.r.t. the specific challenges of RDS, the challenges C1-C3 in particular.

G2 The second goal is to endorse these abstractions at design time by providing appropriate **design models and processes**. The focus here is on formal aspects of these models and processes in order to enable formal model analysis and process automation, while also accounting for design scalability and open-endedness. Thus, this goal primarily relates to the challenges C3-C4.

G3 The last goal, pertaining to the challenges C1-C4, is to detail the **semantics** of the architecture abstractions so that it is suitable for **distributed and decentralized execution** in the dynamic physical environment of RDS and, at the same time, allows for **formal analysis** to ensure dependability. This goal also includes a realization of this detailed semantics in terms of a mapping to a programming language and an execution environment prototype.

1.5 Overview of Contribution

The main contribution presented in this thesis consists of a commented collection of co-authored publications. Most of the results presented in these publications stem from research work and collaboration within the EU FP7 project ASCENS [5]. The results were also applied as a part of the project's industrial demonstrators [SRA+11].

First, in [KBPK12] and [BGH+13] we have proposed architecture abstractions tailored for designing open, inherently dynamic software architectures that adapt based on the current observable system context. The benefit is that these abstractions, the *component ensemble* abstraction in particular, enable declaratively capturing the invariant architecture patterns that recurrently appear in RDS at runtime in a way that governs resilience to extensive architecture dynamism and communication link instability. This in turn makes the runtime state of the dynamic RDS architectures more predictable and dependable. We have summarized these abstractions into the newly introduced *DEECo component model* [8] (Dependable Emergent Ensembles of Components). Moreover, we have equipped DEECo with a rigorously defined computational model, facilitating formal analysis. We have also provided a Java-based execution environment prototype – jDEECo [13], supporting these abstractions at runtime.

Second, in [KBP+13] we have further supported the DEECo abstractions at design time by providing a tailored design method – Invariant Refinement Method (IRM). IRM helps effectively exploiting the DEECo abstractions during design and governs a traceability between system requirements and DEECo-based architectures. The benefit is that this traceability enables design validation and analysis, thus facilitating dependability. IRM builds on goal-oriented requirements elaboration, while integrating aspects of component-based architecture design and software control system design.

Third, in [KBPH14] we have presented ARCAS – a method for open-ended design and automated synthesis of software connectors that is particularly suitable for realization of dynamic, emergent component bindings, which are the cornerstone of RDS architectures. The main contribution lies in the idea of structuring software connectors as hierarchical composites of reusable connector elements and the idea of automated connector composition via constraint solving. This way, while providing proper separation of concerns, ARCAS allows for taking into account factors such as extra-functional requirements on the connectors and capabilities of the heterogeneous deployment nodes.

Fourth, we have focused on the dependability aspects of the DEECo component model and the resulting challenges for verification. Specifically, in [BBB+13] we have presented a formalization of the general operational semantics of DEECo and discussed the opportunities for verification of DEECo-based applications via mapping the semantics onto DCCL [BBCP13]. DCCL is a semantic model embodying abstractions suitable for model checking of ensemble-based systems, introducing, however, additional abstraction and simplification. On one hand, we have identified a list of properties observable under the DEECo semantics that can be verified using the simplified model-checking semantics. On the other hand, we have identified the limitations of the model-checking semantics both in terms of expressiveness, as it introduces additional abstraction, and performance, as the highly concurrent and dynamic DEECo-based systems generate a very large state space.

Fifth, we have focused on application of DEECo in relevant problem domains. Specifically, in [BBHK13] we have proposed a DEECo-based architecture governing adaptive deployment in ad-hoc cloud systems. In [MKH+13] we have put the ideas presented in [BBHK13] into the context of voluntary cloud computing researched within the ASCENS project. Moreover, in cooperation with Volkswagen AG, we have elaborated and implemented several variants of the cooperative vehicle navigation scenario [SRA+11] featured by the ASCENS project; the results, however, are not publicly available as they fall under a non-disclosure agreement. An excerpt of these results can be found in [SHP+13, SMP+12]. In all these cases, the experiments showed that the DEECo architecture abstractions are advantageous and effectively address the challenges of dynamism, communication-link instability, and open-endedness typical for RDS.

1.6 Publications

The following reviewed publications form the core contribution presented in this thesis. The summaries and full texts of these publications are included in Chapter 3.

[KBPK12]	J. Keznikl, T. Bureš, F. Plášil, and M. Kit. *Towards Dependable Emergent Ensembles of Components: The DEECo Component Model.* In WICSA/ECSA ’12: Proceedings of the Joint 10th Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture, pages 249–252. IEEE CS, August 2012.
[BGH+13]	T. Bureš, I. Gerostathopoulos, P. Hnětynka, J. Keznikl, M. Kit, and F. Plášil. *DEECo: an Ensemble-Based Component System.* In CBSE ’13: Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering, pages 81–90. ACM, June 2013.
[KBP+13]	J. Keznikl, T. Bureš, F. Plášil, I. Gerostathopoulos, P. Hnětynka, and N. Hoch. *Design of Ensemble-Based Component Systems by Invariant Refinement.* In CBSE ’13: Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering, pages 91–100. ACM, June 2013. Awarded with the Distinguished Paper Award.
[KBPH14]	J. Keznikl, T. Bureš, F. Plášil, and P. Hnětynka. *Automated resolution of connector architectures using constraint solving (ARCAS method).* Software & Systems Modeling, 13(2):843–872. Springer Berlin Heidelberg, May 2014.
[BBB+13]	J. Barnat, N. Beneš, T. Bureš, I. Černá, J. Keznikl, and F. Plášil. *Towards Verification of Ensemble-Based Component Systems.* In FACS ’13: Proceedings of the 10th International Symposium on Formal Aspects of Component Software, volume 8348 of Lecture Notes in Computer Science. Springer, October 2013. In press.
[BBHK13]	L. Bulej, T. Bureš, V. Horký, and J. Keznikl. *Adaptive Deployment in Ad-Hoc Systems Using Emergent Component Ensembles: Vision Paper.* In ICPE ’13: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering, pages 343–346. ACM, April 2013.

The publications [BGH+13] and [KBPH14] are of equal authorship. In [KBPK12] and [BBHK13], under helpful guidance and supervision of the other authors, I came up with the main idea and authored most of the text. In [KBP+13] and [BBB+13], again under helpful guidance and supervision of the other authors, I authored a majority of the text. Additionally, in [KBP+13] I contributed by elaboration and formalization of the main idea, while in [BBB+13] I elaborated the main idea and conducted the case study.

A summary of the contributions discussed in this thesis was also presented as the following reviewed poster publication, which is of equal authorship.

[AABG+14a] R. Al Ali, T. Bureš, I. Gerostathopoulos, P. Hnětynka, J. Keznikl, M. Kit, and F. Plášil. *DEECo: an Ecosystem for Cyber-Physical Systems*. In ICSE '14: Companion Proceedings of the 36th International Conference on Software Engineering. ACM, June 2014. Poster and extended abstract.

In addition, the following co-authored reviewed publications support the contributions listed above by sharing the underlying topics of (component-based) software architecture and software adaptation.

[BGH+14a] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil. *Gossiping Components for Cyber-Physical Systems.* In ECSA '14: Proceedings of the 8th European Conference on Software Architecture. Springer, August 2014. Accepted for publication.

[AABG+14b] R. Al Ali, T. Bureš, I. Gerostathopoulos, J. Keznikl, and F. Plášil. *Architecture Adaptation Based on Belief Inaccuracy Estimation*. In WICSA '14: Proceedings of the 11th Working IEEE/IFIP Conference on Software Architecture. IEEE CS, April 2014.

[BBK14] M. Babka, T. Balyo, and J. Keznikl. *Solving SMT Problems with a Costly Decision Procedure by Finding Minimum Satisfying Assignments of Boolean Formulas.* In R. Lee, editor, Software Engineering Research, Management and Applications, volume 496 of Studies in Computational Intelligence, pages 231–246. Springer International Publishing, 2014.

[SBK13] N. Serbedzija, T. Bureš, and J. Keznikl. *Engineering Autonomous Systems.* In PCI '13: Proceedings of the 17th Panhellenic Conference on Informatics, pages 128–135. ACM, September 2013.

[MKH+13] P. Mayer, A. Klarl, R. Hennicker, M. Puviani, F. Tiezzi, R. Pugliese, J. Keznikl, and T. Bureš. *The Autonomic Cloud: A Vision of Voluntary, Peer-2-Peer Cloud Computing.* In SASOW '13: Proceedings of the IEEE 7th International Conference on Self-Adaptation and Self-Organizing Systems Workshops, pages 89 – 94. IEEE CS, September 2013.

[BBH+12] L. Bulej, T. Bureš, V. Horký, J. Keznikl, and P. Tůma. *Performance Awareness in Component Systems: Vision Paper*. In COMPSACW '12: Proceedings of the 36th IEEE Annual Computer Software and Applications Conference Workshops, pages 514–519. IEEE CS, July 2012.

[BBK+12] L. Bulej, T. Bureš, J. Keznikl, A. Koubková, A. Podzimek, and P. Tůma. *Capturing Performance Assumptions Using Stochastic Performance Logic.* In ICPE '12: Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering, pages 311–322. ACM, April 2012.

[KMBH11] J. Keznikl, M. Malohlava, T. Bureš, and P. Hnětynka. *Extensible Polyglot Programming Support in Existing Component Frameworks.* In SEAA ’11: Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications, pages 107–115. IEEE CS, August 2011.

[PKH+11] T. Pop, J. Keznikl, P. Hosek, M. Malohlava, T. Bureš, and P. Hnětynka. *Introducing support for embedded and real-time devices into existing hierarchical component system: Lessons learned*. In SERA ’11: Proceedings of the 9th International Conference on Software Engineering Research, Management and Applications, pages 3–11. IEEE CS, August 2011.

1.7 Structure

The thesis is divided into two main parts. First, Chapter 2 presents the state of the art in design and analysis of dynamic software architectures of systems akin to RDS, with detailed focus on the research goals **G1**-**G3**. In particular, Section 2.1 overviews the software architecture abstractions, design models, and software-engineering processes that can be potentially beneficial for building RDS. Further, while emphasizing predictability and dependability, Section 2.2 outlines the approaches to architecture (self-) adaptation that provide the driving force of contemporary dynamic software architectures. Finally, Section 2.3 discusses the principal approaches to formal specification and analysis of software architectures with the aim of providing a foundation for addressing the challenge of dependability in scope of the outlined research goals. The second part, Chapter 3, includes a commented collection of six co-authored publications that contribute to the goals **G1**-**G3**. Chapter 4 then concludes the thesis and gives the author’s subjective vision of the promising research directions related to the area of dynamic software architectures for RDS.

Chapter 2

State of the Art

This chapter includes an overview of the state-of-the-art approaches related to dynamic software architectures of RDS in scope of the research goals **G1**-**G3**.

In general, dynamic software architectures represent a very popular research topic, which by itself includes an overwhelming amount of work from different software engineering disciplines. Consequently, as hinted in Section 1.2, the current state of the art approaches already (at least partially) address the specific challenges of RDS discussed in Section 1.3. However, because the area of RDS has emerged only recently, it is difficult to employ these approaches directly, as they do not address all the challenges together. Therefore, the objective of this chapter is to explore the related state-of-the-art approaches and identify the important points that can be advantageously employed for building dynamic architectures for RDS. Specifically, the focus is on software architecture abstractions, their implementations, and corresponding software-engineering methods and processes that would address the requirements of distribution, resilience, scalability, dynamicity, adaptability, and dependability.

Essentially, since this thesis focuses on various aspects of dynamic software architectures for RDS, the state of the art can be categorized into three (partially crosscutting) topics:

- *Software architecture abstractions.* The key aspect of architecture design is the way software parts are structured and organized within the overall architecture. This is largely determined by the employed software architecture abstractions. Contemporary software architectures of large-scale distributed systems akin to RDS employ a large variety of abstractions, each providing different properties and following different objectives. To this end, this thesis focuses on the prevalent idea of organizing large-scale distributed software architectures via a composition of well-defined software parts, which is the common foundation of many component and service-oriented approaches. In particular, Section 2.1 overviews the approaches focusing on component-based architectures, service-oriented architectures, agent-based architectures, and service-component ensembles.

- *Dependable software architecture (self-) adaptation.* Orthogonally to the architecture abstractions and their implementations, various software architecture (self-) adaptation mechanisms have been proposed so as to provide the driving force of the contemporary dynamic software architectures. These approaches bring different tradeoffs between the scope of the adaptation and architecture predictability, dependability, and scalability and, therefore, offer variable utility regarding the specific needs of RDS. Adopting a view on architecture dynamism that is orthogonal to Section 2.1, Section 2.2 overviews architecture (self-) adaptation approaches ranging from virtually unrestricted ad-hoc (self-) adaptation, through various kinds of restricted (self-) adaptation and autonomic (self-) adaptation, to inherently dynamic architectures.

- *Formal methods in (dynamic) software architectures.* In order to ensure dependability of (dynamic) software architectures, which is one of the key concerns present throughout the goals of this thesis, a significant body of work focuses on application of formal methods. To this end, Section 2.3 presents an overview of approaches to formal specification and analysis of (dynamic) software architectures. In particular, the overview includes approaches focusing solely on structural aspects of (dynamic) architectures, as well as approaches considering also the behaviors governed by the architectures.

2.1 Software Architecture Abstractions

In this section, we overview the approaches to software architectures in distributed systems, including the related software architecture abstractions, models, and software-engineering processes, that are based on the prevalent concept of organizing the architectures via a composition of well-defined software parts – components. As already indicated in Section 1.2, although its primary goal is decomposition and separation of concerns, the component abstraction brings many other software-architecture properties, which are potentially beneficial for building RDS.

To this end, as a baseline for the work presented in this thesis, we first overview the approaches to component-based software architectures, which target architecture-design scalability via (de)composition and reuse (Section 2.1.1). As is frequently the case in today’s industrial applications, this also covers architecture design for embedded and real-time systems, which is an important aspect of RDS. Focusing on the component communication and coordination concerns, especially w.r.t. distribution, we then provide a discussion of software connectors (Section 2.1.2). Further, we investigate the specifics of service-based systems, which represent a conceptual shift towards loosely-coupled, open-ended, and dynamically-composed software architectures (Section 2.1.3). To respond the need for autonomy in RDS, we also overview architecture abstractions based on the concept of software agents, which bring the autonomy to architecture organization (Section 2.1.4). Reflecting the recent progress in software architectures towards a synergy of components, services, and agents, we also elaborate on the state-of-the-art results based on the concept of service component ensembles (Section 2.1.5). Finally, we conclude this section with a discussion of how the presented architecture abstractions can be used for building dynamic architectures for RDS (Section 2.1.6).

2.1.1 Component-Based Architectures

Component-Based Development (CBD) [CL02, HC01, Szy02] is a widely adopted approach to software architecture design. It relies on separation of concerns to tame the complexity of building and maintaining large applications [CL02]. Its main benefit is that it provides abstractions for software (de)composition, interoperability, and reuse.

CBD has been adopted in many diverse domains including configuration platforms (e.g., OSGi [HPMS11][21], Google Guice [12], Spring [22]), user interface frameworks (e.g., WPF [Fre10], JavaBeans [20]), enterprise applications (e.g., EJB [19], CCM [WSO01][18], Sofa 2 [BHP06]), systems software (e.g., OpenCOM [CBG+08]), grid systems (e.g., GCM [BHR14]), and embedded systems (e.g., Koala [VOVDLKM00], MyCCM-HI [BHP09], ProCom [SVB+08], Sofa HI [PWT+08]).

The key notion of CBD is a *component model*, which defines (and gives semantic meaning to) the *component* architecture abstraction and the mechanism of component *composition*. In principle, a component is a software unit with contractually specified interfaces and explicit context dependencies, which can be developed and deployed independently and is subject to (third-party) composition [Szy02]. Due to the critical role of the component model, a component can also be seen as a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard [HC01].

At runtime, components rely on the services provided by an *execution environment* (also referred as component platform). Essentially, an execution environment implements the corresponding component model. The joint product of a component model, its execution environment, and the related tools (including tools for design, development, and deployment), is frequently referred to as a *component system* [CL02].

2.1.1.1 Component Model

In general, a component model focuses on the concerns related to component interface, composition, implementation, interaction, packaging, deployment, and execution.

The interface-related concerns determine how a component's provisions/requirements are specified. For example, while OSGi [21] specifies the provisions/requirements in terms of procedural provided/required interfaces, ProCom [SVB+08] employs the concepts of data ports and triggering ports.

The composition-related concerns determine the support of component nesting, the component abstraction level, and the way component bindings are captured and established. For example, many component models, such as Sofa 2 [BHP06], Koala [VOVDLKM00], and Fractal [BCL+06], allow for hierarchical component composition (i.e., a component can consist of a composition of sub-components). Hierarchical composition also often relates to elaborate component abstractions. For instance, Sofa 2 [BHP06] provides the abstractions of (i) component type, describing the outer contract of a component (i.e., black-box view), (ii) component architecture, describing the internal structure of a component (i.e., gray-box view), and (iii) component instance, representing a concrete instantiation of a component architecture within component hierarchy. On the other side of the spectrum, component models such as OSGi [21] do not support component nesting and perceive components as singletons (without any additional component abstractions). Also, while in the vast majority of component models the component bindings are specified explicitly by the architecture, service-oriented component models, such as OSGi [21] and iPOJO [EHL07], describe component bindings implicitly by specifying the component-interface types and rules for their interconnection. (Service-oriented architectures are discussed separately in Section 2.1.3.)

The implementation-related concerns determine the way a component is defined and implemented. For example, in the Java-based component platform of Fractal – Julia [BCL+06] – a component can be implemented, among other options, in terms of a specifically annotated Java class or in terms of an abstract composite defined via a dedicated Architecture Definition Language (ADL).

The interaction-related concerns determine the communication styles based on which components interact. For example, while Koala [VOVDLKM00] relies solely on synchronous method call, EJB [19] and CCM [18] also allow for asynchronous messaging. Moreover, some component models, such as Sofa 2 [BHP06], Wright [AG97], and Acme [GMW00], employ the concept of *software connectors* [MMP00] that encapsulates and hides the specifics of a particular communication style. (Software connectors are further discussed in Section 2.1.2.)

The concerns related to packaging and deployment determine how components are treated during deployment. For example, components in OSGi [21] are packaged and deployed as standalone JAR archives, whereas Sofa 2 [BHP06] provides a dedicated deployment infrastructure relying on a component repository.

Finally, the execution-related concerns determine the programming model of a component's internal behavior and the contract w.r.t. to the component's execution environment. For example, component models targeting real-time systems, such as MyCCM-HI [BHP09], Sofa HI [PWT+08], and ProCom [SVB+08], ensure real-time scheduling of component behaviors. In a similar vein, GCM [BCD+09, BHR14] provides mechanisms for synchronization of asynchronous component execution in grid systems.

Naturally, there is a large variety of distinct component models. Though different in many aspects, all the component models share the fundamental idea of components as blocks of functionality with relatively well-defined architecture and interaction patterns. A comprehensive survey can be found for example in [LW07], [CVZ+11], [HPB+10], and [PHH+13].

2.1.1.2 Execution Environment

At runtime, components are executed within an execution environment, which takes care of their lifecycle, binding, interaction, and various non-functional concerns, such as introspection and concurrency [CL02]. Moreover, the execution environment needs to take care of all the specific features of the corresponding component model. Depending on the actual component model, this might include, for instance, real-time scheduling (e.g., by employing scheduling capabilities of the underlying operating system), mediating distributed communication (e.g., by employing suitable middleware), etc. Conversely, a component model can reflect certain specific services of its execution environment (e.g., introspection). For example, the Fractal component model [BCL+06] includes the concept of component membrane, which exposes the capabilities of the Fractal execution environment. As an aside, an execution environment itself may be constructed as a component-based system. For example, the execution environment of Sofa 2 [BHP06] is partially based on a designated "micro-component" model [BHM09]. Similarly, the concept of software product lines [CN02, GS03, PBVDL05] may be employed for producing families of component platforms tailored to various target application domains [BHM09, Mal12].

Execution environment is closely connected to deployment. In fact, it is at the deployment stage of the CBD process [CCL06] when components, as well as the execution environment, are being prepared for execution according to a system architecture. This may include also synthesizing significant parts of the execution environment, such as infrastructure ensuring component distribution [BP04, Bur06]. In principle, there are two types of deployment: compile-time deployment and runtime deployment. During the compile-time deployment, components and the execution environment are built and linked together prior to execution. While bringing predictability and dependability, it prevents changes in the architecture at runtime. (The architecture may still display limited dynamic capabilities, as discussed in Section 2.2.2.) This type of deployment is typical for embedded and real-time systems, running in constrained environments (e.g., MyCCM-HI [BHP09], Koala [VOVDLKM00], PECOS [GCW+02], ProCom [SVB+08]). On the other hand, the runtime deployment enables performing explicit changes to the architecture during execution. Thus, it is critical for enabling architecture adaptation. This type of deployment is advantageously employed in enterprise and pervasive systems featuring continuous execution (e.g., EJB [19], Sofa 2 [BHP06], Fractal [BCL+06], CCM [18]).

2.1.1.3 Component-Based Architectures at Design Time

CBD introduces a specific architecture-design process that is based on the distinct life cycle of software components [CCL06, CL02]. In particular, CBD focuses on building systems by reusing already existing components. Therefore, the CBD development process separates system development (i.e., development of systems by composition of components) from component development (i.e., development of components).

Several component development processes may be on course simultaneously, thus making it possible to develop several components at the same time. Consequently, since the system development and component development are always to some extent parallel processes, CBD puts a significant focus on component interoperability, composability, and validation/verification thereof (verification is further discussed in Section 2.3). Note that the interoperability and composability are to a major extent addressed by proper selection of the underlying component model.

From the perspective of architecture design, mainly focusing on the system development, there are several approaches to the CBD process [CCL06, CL02]. The objective of all these approaches is to determine the component instantiation, composition, and interaction in a component-based software architecture.

The architecture-driven CBD process uses a top-down approach to component-based system design. Specifically, components are identified during high-level analysis and architecture design, e.g., by elaboration of system requirements [YLL+08], so as to capture the essential architecture elements and their responsibilities. Such components are thus rather specific to the architecture, with the focus on composability, and are not primarily designed for reuse. The corresponding component model acts in this case as a common design platform.

The CBD process based on software product lines [CN02, GS03, PBVDL05] focuses on systematic component reuse to produce a desired product family. The purpose of a product line is to identify the common features and variation points of the desired product family in advance. This is achieved by a combination of top-down and bottom-up design approaches. The common features are embodied in a reference/platform architecture, while the variation points are reflected via substitutable components. Compared to the architecture-driven CBD process, the product-line-based approach emphasizes parallelism of the component development and system (i.e., product) development. Here, component model is not only the common design platform facilitating component reuse and integration, but it also reflects the specifics of the particular product family. As a result, there are component models specifically developed for particular product families (e.g., Koala [VOVDLKM00]).

Finally, the CBD process based on third-party components focuses on component reusability, separating component development and system development completely. It relies primarily on bottom-up component design, thus making the component model largely responsible for ensuring composability and interoperability [dJ09].

2.1.1.4 Embedded and Real-Time Component Systems

As already hinted above, one of the domains in which CBD has been widely adopted is the domain of real-time embedded systems (RTES) [But11, CL02]. Examples of embedded and real-time component systems are numerous, including MyCCM-HI [BHP09], Koala [VOVDLKM00], PECOS [GCW+02], ProCom [SVB+08], SOFA HI [PWT+08], and BlueArx [KRKH09].

Despite following the main principles of CBD, such as reusability, compositionality, well-defined architecture design, support for distributed communication, etc., component-based design and development of RTES introduces several specifics [PKH+11]. These specifics are driven by the close interaction of RTES with their ever-changing physical environment. Adopting the approach of control engineering, RTES achieve robustness w.r.t. the recurrent changes in their environment by maintaining *operational normalcy*. Here, operational normalcy expresses the property of being within certain limits that define the range of normal functioning of a system. Technically, this is achieved by adequate scheduling of periodic or event-driven tasks representing sensors acquisition, control, and actuation, all performed in "real-time". In this context, the embedded and real-time component systems provide specific support for periodic and aperiodic tasks within components, their composition, modeling of their real-time attributes, as well as real-time task scheduling at run-time [But11]. Furthermore, as RTESs are often deployed to hardware with limited resources (e.g., low amount of available memory, low-frequency CPU with no memory management unit), the system demands of the corresponding component platforms (e.g., memory footprint) need to be analyzable and configurable.

Another important aspect of RTES is that there is a very strong emphasis on dependability, which brings the need for analysis and verification. The basis for this is twofold. First, RTESs are commonly used in safety-critical applications, such as car systems. Second, as updates of software in embedded devices can be complicated or even impossible, it is desirable to ensure system validity during design. Although this includes also specification and verification of system behavior (discussed in Section 2.3.2), as in the traditional CBD, the primary focus is on analysis and verification of real-time properties, especially w.r.t. composition. The main representatives are (compositional) WCET (Worst-Case Execution Time) analysis [LCS13] and schedulability analysis [But11, HZPK07].

To further promote dependability, embedded and real-time component systems frequently employ model-driven engineering (MDE) and code generation [CFMTS10, PKH+11]. The reason is that models facilitate analysis, while code generation allows retaining the analyzed models' properties in the implementation [BC11]. Also, the generated code itself can be made more suitable for analysis and verification and with more configurable and predictable resource demands than a hand written code. For example, in MyCCM-HI [BHP09] the component-based system design is transformed to the lower-level Architecture Analysis and Design Language (AADL) [FGH06, VPK05]. Then, the corresponding Ocarina tool suite [HZPK07, HZPK08] can be used to produce executable code or to mediate formal verification of behavior and schedulability analysis. Another advantage of code generation is that the generated code can cover the OS and platform-specific aspects, which are not covered by the abstraction layer of the execution environment, and thus can help keeping the component implementation platform-independent. Technically, code generation is typically employed for synthesis of component interconnections (i.e., component connectors, see Section 2.1.2) and the component skeleton code, which is primarily responsible for handling real-time aspects of execution, such as (periodic) real-time scheduling or handling of missed deadlines.

ProCom [SVB+08] is an example of an embedded and real-time component system, which gives a strong emphasis on formal analysis and verification [LCS13, VSC+09, VSCS10], as well as model-driven engineering and code generation [BC11]. The ProCom component model distinguishes two levels of abstraction – ProSys and ProSave. ProSave represents the lower level, focusing on passive, hierarchically structured components communicating via pipe-and-filter style on top of the components' data ports and triggering ports. ProSys, on the other hand, focuses on a description of potentially distributed and concurrent (event-driven or periodic) subsystems communicating via asynchronous message channels. A ProSys component can be represented as an assembly of ProSave components.

Another example of a component system for embedded devices is Koala [VOVDLKM00], developed by Philips and targeting consumer electronics. The component model supports hierarchical component composition at design time and is strongly oriented on capturing design-time variability in terms of software product lines [dJ09]. The code generation in Koala is employed for generating component connections. The generated code, as well as the execution platform, is strongly optimized w.r.t. resource demands. However, Koala does not address real-time execution aspects.

Other examples are surveyed in [HPB+10, PHH+13].

2.1.1.5 Dynamic Component-Based Architectures

As outlined in Section 2.1.1.3, component-based software architectures are at design time eventually captured in terms fully elaborated component compositions. This means that component instantiation, composition, and interaction are typically fully determined at design time. Consequently, this prevents CBD to address the concerns of architecture dynamism directly. To enable dynamic architectures, component systems mostly exploit the runtime deployment support of their execution environments, rather than reflecting architecture dynamism explicitly in the component model.

Nevertheless, some component systems enable capturing dynamic integration and reconfiguration of components at the level of component model so as to cope with dynamic, open-ended environments, while allowing for well-defined component-based architecture design. For example, the AmbiComp approach [PS08] is based on dedicated component controllers that enable ad-hoc instantiation of various pre-designed application scenarios based on the current context of the deployed components. At runtime, the controllers mediate ad-hoc component discovery, negotiation of component provisions, requirements, and extra-functional properties, as well as instantiation of appropriate application scenarios by proper component composition. Moreover, any change in the established component architecture (e.g., appearance/disappearance of a component, change in a component's provisions/requirements, change in the component's quality attributes, etc.) is reflected by the controllers, thus enabling the components to react accordingly. Some other component systems, such as SOFA 2 [BHP06] and Fractal [BCL+06], provide similar reflective capabilities that enable well-defined runtime architecture reconfiguration at the level of component model. These can be then employed by external adaptation mechanisms to achieve architecture dynamism, as discussed separately in Section 2.2.

2.1.2 Software Connectors

Software connectors are entities encapsulating interaction and coordination among software components, thus separating these concerns from the actual component functionality [MMP00, TMD10]. In particular, connectors (i) ensure distribution of communicating components by encapsulating middleware [BP04] (middleware-based connectors), (ii) provide adaptation in order to achieve middleware-level [IBB11, NTER06] and application-level [CCP11, IST11, SI10] interoperability (adaptors), and (iii) ensure synchronization of component communication [BS08, IST11] (coordinators).

At a more technical level, there are various connector types, based on the way in which they realize the distribution/adaptation/coordination. These types include procedure call, event propagation, data access, linkage, stream, and distributor connectors [MMP00, TMD10], to name just a few. Connectors can also provide additional services concerning extra-functional properties of component interaction, such as persistence, transactions, and encryption [BP04, Bur06].

The concept of software connectors is adopted as a generic abstraction generalizing concrete communication styles by various existing component-based approaches to software architectures, e.g., Wright [AG97], Acme [GMW00], ArchJava [ASCN03], and Sofa 2 [BHP06].

2.1.2.1 Connector Synthesis

The connector development process differs from the CBD process. Although connectors are partially employed during the component/system design, fully specified connectors emerge at the earliest in the component deployment phase. This is due to the strong reliance of a connector's architecture and implementation on its deployment (e.g., on the available middleware). On the other hand, the separation of concerns makes connectors rather independent on component functionality, thus rendering them suitable for automated synthesis. Although there are approaches focusing on explicit connector design in combination with traditional CBD process [RCGT09, RRS+05], special focus has been put on automated synthesis of connectors at deployment time and runtime.

Recently, a particular effort has been invested into research of automated synthesis of connectors that ensures application-layer and middleware-layer interoperability [BP04, Bur06, IBB11, ISJ+09]. This is especially beneficial when connecting independently developed or deployed components. The emergence of such connectors is particularly advantageous at runtime when architecture and deployment reconfiguration takes place (e.g., due to load balancing or mobility). Based on a specification of the connected components and the middleware available for each of these components, the goal is to synthesize (in an automated way) a connector implementation that ensures the components' interoperability. Resembling CBD for connectors, an interesting prospective approach is the synthesis based on automated composition of hierarchical connector architectures from reusable elements, reflecting the available middleware, the desired communication style, and certain extra-functional properties pertaining to communication concerns (e.g., encryption) [BP04, Bur06]. As a specific case, a large body of work has been invested in synthesis of the so-called *emergent connectors* [IBB11, ISJ+09], which is basically a synthesis at runtime with limited (or without) preceding connector design. Specification of such emergent connectors is to be programmatically derived from the specification of the components to be connected.

2.1.3 Components as Services

Service-Oriented Computing (SOC) represents a shift from components to *services* [BL06, HS05, Pap03] as the main building blocks of software architectures. Targeting open and heterogeneous environments, SOC aims at eliminating the issues of strict (design-time and runtime) dependencies and centralized deployment of component-based architectures, while retaining the emphasis on (de)composition and interoperability. Although there are many approaches to SOC (e.g., web services [CNW01]), the approaches that promote synergy of SOC and the component-based architectures are of particular interest.

The core idea of SOC is based on clearly separating the responsibilities of the parties involved in a service-oriented architecture (SOA): a *service provider*, a *service consumer* (end-user or another service), and the *service broker*. Specifically, SOC separates the following concerns: (i) service publication (involving a provider and the broker), (ii) service discovery (involving a consumer and the broker), and (iii) service invocation (involving a producer and a consumer). For this purpose, SOC relies on explicit service specification, describing the functionality provided by a service (e.g., WSDL [CCMW01]). An SOA is hence formed by composing service specifications, which get at runtime concretized to actual services by the service broker. As such, service composition can be realized in many ways, including regular code (e.g., OSGi [21]) and process-based approaches (e.g., WS-BPEL [JEA+07]). A composition of services can also itself implement a service specification, thus enabling service nesting.

This brings about several important properties to a service-oriented software architecture [HS05]. First, services in an SOA are at design time loosely coupled, which facilitates service autonomy, contributes to overall SOA robustness, and promotes independent service development and deployment. Further, due to the emphasis on a thorough, implementation-oblivious service specification, SOA promotes service substitutability and heterogeneity. Finally, relying on a service broker, bindings among services are established late and dynamically, which brings resilience to dynamic availability of services at runtime. These properties enable SOA to become very flexible.

At runtime, an SOA is supported by a (usually centralized) service-oriented platform [WCL+05], which provides an infrastructure for service publication, discovery, and invocation, relying of on a set of standardized protocols (e.g., SOAP, WSDL [WCL+05]). The platform also takes care of tracking dynamic availability of services and ensures notification of the affected service consumers. For example, the OSGi platform [21] provides a dedicated API enabling the service consumers to cope with dynamic availability of service providers.

Nevertheless, SOC also brings limitations w.r.t. highly-dynamic adaptive systems [DNGM+08]. Namely, being conceptually centralized, service-oriented platforms prevent efficient distribution and present a scalability bottleneck for service discovery and binding [AZI09]. Moreover, although supporting dynamic availability, service compositions typically rely on stable communication channels once connected and do not cope well with sudden disconnections.

2.1.3.1 Service Components

In principle, SOC does not primarily address the concerns of complexity and reuse [MR09, YP04]. This stems from the emphasis on loose coupling. Specifically, SOC, and web services in particular, focuses on interaction of pieces of software (services) that have nothing in common, i.e., they form “isolated islands of functionality” that interact with each other. While this can be certainly considered an asset, it limits design scalability and the potential for reuse. To compensate for this, the concept of *service component* has been proposed [CH04].

The service-component approach promotes SOC-based composition while employing CBD for service implementation. In a *service-oriented component model* [CH04, EHL07], the concept of a service acts as a unit of provided/required functionality, characterized by an explicit service specification (including syntactic, behavioral, and semantic concerns, as well as service dependencies). A component acts as an implementation of potentially multiple services and may feature its own implementation-specific service dependencies, i.e., components interact by providing services and using the services of other components. Component composition is described and established upon the components’ services via a combination of the SOC mechanisms and other techniques (e.g., message bus) implemented by the service-component platform. Hierarchical composition is achieved by allowing component compositions to implement further services. Contrary to CBD, service-component instances are not necessarily declared at design time, but emerge dynamically at runtime – either due to explicit activation or due to implicit instantiation via dedicated factories (driven by the runtime platform to satisfy the demand for provided services).

The service-component paradigm has been applied in a number of component frameworks, such as Service Binder [CH04] (now integrated into OSGi [21]) and iPOJO [EHL07]. An interesting feature of Service Binder is that it enables runtime instantiation of complex abstract composites by binding predefined service placeholders to services of the available component instances (based on the service specifications and component-specific properties). Conversely, iPOJO focuses on integrating simple POJO components (Plain Old Java Objects) into a service-oriented framework, complemented by externally-defined service-oriented compositions.

A different approach to service-components is portrayed by Service Component Architecture (SCA) [MR09][17]. In principle, SCA tries to combine explicitly-defined component-based architectures with service-oriented composition [HP06]. The motivation is an integrated composition and deployment of related services, while ensuring loose coupling wr.t. services of third parties. Thus, in addition to adopting the traditional service-component concepts (i.e., components as implementations of well-defined services), SCA enables explicit (in the CBD sense) composition, instantiation, and deployment of components implementing related services. In principle, SCA is a technology for assembling service applications from components that are managed by a common infrastructure. Consequently, SCA focuses on the concerns of portability and reusability of heterogeneous service components developed by different parties, on different platforms, using different communication protocols and paradigms. On the other hand, it leaves the other concerns, such as interoperability and runtime management, on external mechanisms (e.g., interoperability via web services) and the particular implementations (e.g., FraSCAti [SMR+12][10] and Tuscany [4]). An in-depth comparison of SCA with a fully-fledged component model – Fractal [BCL+06] – is presented in [HMM11].

2.1.4 Components as Software Agents

To mitigate the conceptual reliance of components/services on each other, Agent-Oriented Computing (AOC) has introduced *software agent* as an abstraction that brings conceptual autonomy to loosely coupled system components. An agent can be perceived as a software abstraction that exhibits some degree of autonomic behavior and is capable to cooperate in order to achieve both individual and collective goals. Each agent is designed to operate with a partial view of the whole system, which is beneficial when the global state is not available. For example, in the Belief-Desire-Intention (BDI) architectural model [RG+95] the agents maintain a belief about the rest of the system to guide their autonomous reasoning.

At the level of software architecture, agents are usually implemented and composed via service-component mechanisms, while agent interaction is realized via targeted messaging [BPR01]. Building on the service-composition principles, *multi-agent systems* [SLB09] feature the concepts of agent roles and groups [FGM04]. By extending the idea of abstract service-component composites, the concepts of roles and groups bring the autonomy to architecture organization and allow building self-organized systems that are heavily dynamic. In principle, each agent autonomously decides, based on its individual capabilities, which communication groups it will participate in and under which role. The interaction of agents in a group is then determined by the respective roles of the agents. These concepts are supported at runtime by the corresponding agent platforms, such as JACK [BRHL99] and JADE [BPR01].

As to agent-oriented component-based architectures, approaches such as MetaSelf [DMSFR10] combine self-organization policies evaluated at runtime, agent-inspired autonomy, and rule-based reasoning with a service-oriented architectural framework that is centered on the concepts of self-describing service components. Other approaches, such as the Accord framework [LPH04], focus on service-component autonomy achieved via smart, agent-based composition infrastructure. In particular, Accord assigns to every autonomous component a set of composition rules (corresponding to roles) derived from workflows (corresponding to groups) for driving composition-related aspects in a decentralized manner (including configuration, interaction and coordination), while separating these aspects from computation behaviors provided by the autonomous components.

AOC brings also specific design methods that are tailored to the autonomy and dynamism of agents. In particular, Tropos [BPG+04] is a methodology for designing agent-oriented software systems based on the Goal-Oriented Requirements Engineering (GORE) [RBAF10, VL01], thus aligning requirements analysis with system design and implementation. Tropos is based on the idea of using GORE notions (actors, goals, plans) in all software development phases: from early requirements, through design, down to actual implementation. Specifically, based on the system requirements, Tropos captures the relevant system actors and their interdependencies in terms of goals to be achieved. This includes a specification of how the goals can be fulfilled through contributions of the individual actors (by executing tasks). Such specification is then refined in terms of independent sub-systems that are interconnected through data and control flows and composed of actual software agents with their individual capabilities. This refinement then directly leads to a detailed design of the agents' behavior and interaction protocols that can be readily mapped to a concrete implementation on top of an agent platform (e.g., JACK [BRHL99]). Similar approaches have been adopted also for design of (not necessarily agent-oriented) adaptive architectures [MPP08, TPYZ09, YLL+08].

To summarize, AOC delivers at the conceptual level excellent means for implementing large-scale distributed systems featuring advanced autonomic behaviors. Compared to traditional component-based and service-component approaches, adaptation and dynamism are easier to achieve, since agents are more autonomous.

A problem arises at the software-engineering level. Specifically, agent-oriented approaches, as well as the agent-based platforms (e.g., JACK [BRHL99] and JADE [BPR01]), do not translate the conceptual autonomy and other useful agent notions (i.e., goals, intentions, roles, groups) into proper software engineering constructs. The reason is that they are usually implemented via service-oriented mechanisms, thus sharing the limitations. This includes reliance on an explicit (message-oriented) communication, a (centralized) agent platform, and reliable communication channels between the agents.

2.1.5 Service Component Ensembles

Extending the focus of service components (i.e., open-ended system design, dynamic service availability, design scalability, and reusability) with the concerns of (i) efficient design and distributed execution on a large scale, (ii) (self-) adaptation to changing requirements and non-deterministic environments, and (iii) resilience to node failures, the concept *of service-component ensembles* (SCEs) [HRW08] has recently gained attention and been investigated in the scope of the FP7 project ASCENS [5].

The objective is to build systems from dynamic, self-organizing groups – ensembles – of self-aware, adaptive service components while controlling and engineering their emergent behavior so as to facilitate dependability.

The underlying semantics of SCEs is captured by the SCEL coordination language [DNFLP13, DNLPT14] (based on KLAIM [DNFP98]) where *attribute-based communication* is employed to model a context-driven, dynamic interaction of components. Specifically, the target of communication is determined according to the attributes of both the source and target (rather than by a direct identifier of the target). In terms of SOC, this corresponds to promoting the mechanism of attribute-based service discovery from service composition stage to individual service interaction, while extending the discovery to consider the attributes of both the target and source of the communication (and their mutual relation). The service components in SCEs are portrayed as autonomous, self-aware, and self-adaptive entities, closely resembling software agents.

Addressing, among others, the challenges of security and trust, the SCE paradigm also allows for reflecting various interaction policies [MPT13], which are crucial when building open, heterogeneous systems.

The SCE paradigm can be advantageously exploited to model a dynamic coordination of heterogeneous components on a large scale; moreover, it allows for dealing with dynamic and open environments, as well as with limited reachability of components. A particular benefit is that the concepts of SCEs are formally grounded due to their origins in the SCEL specification language. Therefore, they are equipped with precise operational semantics that provides opportunities for formal analysis. (The formal aspects are discussed separately in Section 2.3.)

In its current form, however, the SCE paradigm relies on strong consistency among the components, which is not plausible in large, decentralized systems. Besides, the current body of work still falls short in providing appropriate software architecture abstractions. Namely, the SCEs are currently not reflected at the architecture level but are embedded within the low-level interaction logic of the components. Consequently, there is a lack of suitable abstractions and software-engineering processes that would support architecture design of SCEs. Also, being relatively novel, the concepts of component ensembles and attribute-based communication are not yet fully explored with regards to appropriate programming models and runtime environments that would enable their efficient implementation and distributed execution. To this end, jRESP [DNLPT14][14], a faithful implementation of the SCEL operational semantics, represents a promising step in this direction.

2.1.6 Lessons Learned

To provide a foundation for pursuing the research goals **G1**-**G3**, the previous sections have discussed several principal approaches towards software architectures in distributed systems akin to RDS. These approaches employ various software architecture abstractions displaying distinct benefits and limitations. The discussion has revealed that there is no readily-available solution to the challenges of RDS (i.e., challenges C1-C4) at the level of software architecture. Nevertheless, the majority of the discussed approaches provide valuable partial solutions.

Component-based development (CBD), discussed in Section 2.1.1, represents a solid baseline for engineering well-defined software architectures for RDS. Specifically, structuring software architectures in terms of well-encapsulated, composable, reusable, and substitutable components enables addressing the challenges of scalability, while providing a base for ensuring dependability (as further discussed in Section 2.3). CBD also facilitates separation of behavior and interaction concerns, which is critical for efficiently addressing the communication-related challenges in RDS. Furthermore, CBD in the area of embedded and real-time systems provides mature techniques for achieving robustness when interacting with the physical environment. In particular, the important idea is to maintain operational normalcy by adequate real-time scheduling of periodic or event-driven component tasks. Finally, CBD is equipped with various top-down and bottom-up design processes, supporting component-based architectures at design time. Nevertheless, CBD does not explicitly account for architecture adaptation and open-ended architecture design. Neither does it cope well with communication and component failures.

CBD can be advantageously supplemented with the concept of software connectors, discussed in Section 2.1.2. In addition to separation of concerns, software connectors bring about the possibility of automated synthesis of communication infrastructure in face of heterogeneous deployments and architecture dynamism of RDS. However, combining dynamic, open-ended architectures with connector synthesis is still an issue.

Certain drawbacks of CBD are addressed by service-oriented component models in the context of service-oriented computing (SOC), discussed in Section 2.1.3. In general, SOC provides the means for dynamic architecture adaptation in face of changing availability of services/components and enables loose service/component coupling and open-ended architecture design. Specifically, SOC introduces the concept of implicit connections based on design-time service specification and runtime service discovery and binding. Nevertheless, in addition to not being intended for close integration with the physical environment, SOC still falls short in addressing the challenges of frequent communication and service/component failures. Also, it is challenging to achieve fully decentralized operation of service-oriented architectures.

Agent-oriented computing (AOC), discussed in Section 2.1.4, well complements CBD and SOC by focusing (at the conceptual level) on autonomous operation and dynamic organization of software agents based on the agents' partial view (belief) on the whole system. Therefore, it facilitates resilience to agent failures and enables efficient decentralized execution. This makes AOC very appealing for RDS. It is also equipped with design methods based on goal-oriented requirements engineering that make designing such decentralized, dynamic, and autonomic systems feasible. Nevertheless, AOC is concerned primarily with taming the autonomic agents' behaviors, rather than with the challenges pertaining to software architecture. In particular, AOC is at the technical level based on service-oriented mechanisms and thus still fails to address the challenge of frequent communication errors in RDS.

Building on the ideas of SOC and AOC, service component ensembles (SCEs), discussed in Section 2.1.5, present a promising approach towards dealing with very dynamic and open environments on a large scale, as well as with limited reachability of components (including frequent communication errors). This is largely due to the related mechanism of attribute-based communication. Moreover, the concepts of SCEs are formally grounded and thus provide opportunities for ensuring dependability via formal analysis (discussed separately in Section 2.3). However, the issues of appropriate software architecture abstractions, implementation, and design processes for SCEs are still not yet fully investigated. To this end, due to the origins of SCEs in service-component architectures, CBD presents a fitting candidate to draw from.

2.2 Dependable Software Architecture (Self-) Adaptation

In this section, focusing on predictability and dependability, we outline various approaches to architecture (self-) adaptation that provide the driving force of contemporary dynamic software architectures. Specifically, we do it in a way that is to some extent orthogonal to the architecture abstractions discussed in Section 2.1. The reason for this is that the core concepts of these approaches are not bound to any specific architecture abstraction (even though some are more related). Here, adaptation means an arbitrary change of software architecture at runtime (a.k.a. reconfiguration), including addition/removal of a component/service, change in component/service composition, change of component/service configuration (if applicable), etc., whereas self-adaptation means adaptation performed by the system itself in response to changes in its operating context [OHJ+99]. This context may comprise both the external environment and the internal state of the system itself, including both functional and extra-functional aspects (e.g., performance [BBH+12]). We specifically focus on architecture self-adaptation.

The common aspect of the discussed self-adaptation approaches is that they are based on the concept of autonomic feedback loop [BSG+09, KC03, OHJ+99], taken from the area of control engineering [MPS08, PCHW12]. Specifically, self-adaptation includes monitoring (awareness) of the application context, detection of a situation requiring adaptation, analysis and assessment of a desired adaptation (e.g., based on predefined adaptation tactics), and execution of the adaptation. For the purpose of monitoring and adaptation execution, self-adaptation typically relies on middleware-level or architecture-level reflective capabilities of the execution environment [ADLMW09], which in turn often involves techniques based on models@runtime [BBF09].

The discussed approaches provide different tradeoffs between the scope of architecture dynamism and its predictability, analyzability, and dependability. This means that approaches enabling arbitrary architecture dynamism may suffer from absolute unpredictability and even complete architecture erosion [BHH+06], while maintaining predictability (and thus analyzability and dependability) limits the dynamism.

To this end, we first overview the ad-hoc approaches to architecture (self-) adaptation, featuring maximum adaptation flexibility, but hindering predictability and dependability (Section 2.2.1). Next, we discuss the approaches based on bounded architecture (self-) adaptation, i.e., adaptation that is limited to a bounded number of pre-defined architecture variants – modes; these approaches are primarily aimed at predictability and analyzability (Section 2.2.2). Offering a balanced tradeoff between flexibility and predictability, we further investigate approaches employing unbounded (but constrained) architecture (self-) adaptation (Section 2.2.3). Last but not least, we also overview inherently dynamic software architectures, enabling predictable (self-) adaptation without changes to the architecture design (Section 2.2.4). We conclude this section with an analysis of how the presented approaches can be utilized for building dynamic architectures of RDS (Section 2.2.5).

2.2.1 Ad-hoc (Self-) Adaptation

In the context of this thesis, we use the term "ad-hoc adaptation" for all approaches in which adaptation is not explicitly modeled at the level of software architecture, but is captured at a lower level, e.g., in terms of adaptation API invocations in code.

Ad-hoc (self-) adaptation is frequently employed in combination with traditional CBD. In particular, since component instantiation and composition is typically decided during design or deployment [CCL06], CBD deals mostly with static architectures (as discussed in Section 2.1.1.5). Hence, relying on the runtime-deployment support, architecture self-adaptation is usually handled in an ad-hoc manner via the reconfiguration capabilities of the corresponding component platforms, such as the membrane control API in Fractal [BCL+06, POS06], dedicated component controllers in Sofa 2 [BHP06, KMBH11] and GCM [BCD+09], or the introspection and reconfiguration API in OSGi [21]. In particular, the ad-hoc self-adaptation takes the form of low-level event-condition-action rules that correspond to invocations of a reconfiguration mechanism reacting to distinct (external or internal) changes in the system's context.

However, hardwiring (self-) adaptation logic directly into the business code is not plausible for development of large evolving applications. The reason is that such hardwiring mixes business concerns with adaptation tactics, which makes both initial development and maintenance more difficult. Furthermore, it is hard to update the adaptation tactics when the application logic evolves or new adaptation contexts are identified. To address this issue, approaches such as [DL06] enable developing the (self-) adaptation code separately and then integrating it dynamically into the business code so as to achieve decoupling, both spatial and temporal. In particular, [DL06] employs aspect-oriented programming [KLM+97] to dynamically extend Fractal-based systems with adaptation tactics relying on low-level reconfiguration scripts based on Fractal's introspection/reconfiguration capabilities.

The advantage of this type of (self-) adaptation is that it is extremely flexible, while still enabling modularity and reusability. On the other hand, since the adaptation is not explicitly modeled at the architecture level, it is hard or even impossible to perform analysis and validation. Furthermore, ad-hoc (self-) adaptation does not offer support for determining interactions between different adaptation tactics. Consequently, it can lead to "uncontrolled" architecture modification, which is inherently error-prone and may result in architecture erosion.

2.2.2 Bounded (Self-) Adaptation and Architecture Modes

Complex systems can encounter a large variety of different contexts that require runtime (self-) adaptation towards different architecture configurations. Addressing these globally is not plausible. As a remedy, a widely-adopted approach is to employ so-called architecture *modes* [HKMU06], which are in principle pre-defined architectural variants specific to different situations that are switched at runtime based on the current context.

Mode-based (self-) adaptation can be advantageously modeled at the level of architecture at design time by capturing the context-specific architecture variants in terms of alternative component/service compositions. This strengthens the relation between design-time architecture models and runtime architecture evolution, and thus facilitates design-time analysis. Consequently, mode-based (self-) adaptation prevents architecture erosion and promotes predictability and dependability. On the other hand, it strictly bounds the allowed architecture dynamism to the pre-designed configurations. (Note that this can be complemented by mechanisms for updating modes at runtime [BGF+08].)

At runtime, mode specifications can be either employed directly to drive the (self-) adaptation [FHS+06] (i.e., models@runtime approach) or translated at design/deployment time to low-level adaptation code [BHP09].

2.2.2.1 Modes in Embedded and Real-Time Component Systems

The predictability and bounded scope of mode-based (self-) adaptation is especially utilized in the domain of embedded and real-time component-based architectures (discussed in Section 2.1.1.4). This is mainly because the ad-hoc approach to architecture (self-) adaptation is not viable due to the unpredictable performance and resource demands, which is unacceptable for correct real-time execution and in face of resource limitations.

To achieve the predictability, it is necessary to employ appropriate design-time architecture models and analysis techniques. As to the architecture models, the focus is on precise mode and mode-change specification that would facilitate analysis, as well as efficient execution at runtime. For example, the approach of [YCH12, YQCH13] presents a mode-aware extension to ProCom [SVB+08], focusing on rigorous, logic-based mode specification. Another example represents MyCCM-HI [BHP09, VPK05], which is a component-oriented extension of the AADL design language [FGH06] and which focuses on explicit specification of mode transitions. As to the analysis, a particular challenge in this context is to ensure timely operation when a system is undergoing a mode change [BDFR08, BHP09, RBF+08, YH13]. Another challenge stems from the requirement of compositionality of real-time mode-change mechanisms along (hierarchical) component composition [PPO+12, YQCH13].

Mode-based (self-) adaptation brings also efficiency w.r.t. resource constraints, since it offers the possibility to optimize the mode change mechanisms [RC04] at design time.

At runtime, it is necessary to introduce explicit support for real-time mode-based (self-) adaptation into the execution environment, as, e.g., in MyCCM-HI [BHP09] and ProCom [YQCH13]. A comprehensive overview of component-based frameworks that, to some extent, provide such support is given in [HPB+10, PHH+13].

Due to the requirements of predictability and resource efficiency, some embedded/real-time component systems do not employ explicit representation of modes at runtime; instead, mode specifications are at design time (e.g., in BlueArX [KRKH09]) or deployment time (e.g., in MyCCM-HI [BHP09]) transformed to an efficient, platform-specific representation.

2.2.2.2 Modes as Composites

In the cases where architecture (self-) adaptation involves variability w.r.t. a number of features, treating modes as homogeneous architecture configurations introduces an explosion of the number of potential modes, which hinders development and maintenance. Moreover, because such features are often partially independent, there is an explosion of the (implicit) transitions between the modes.

To address this issue, the concepts of software product-lines [CN02, GS03, PBVDL05] can be advantageously used for modeling the modes along the individual variability points [BGF+08, FHS+06]. This is usually done by providing different variants of individual system (sub-) components for different contexts so that, at runtime, the architecture mode pertaining to the current context is dynamically composed from the corresponding variants. For example, in the hierarchical component composition of Madam [FHS+06], each sub-component may have multiple implementation variants. Each variant is associated with a utility function that evaluates the variant w.r.t. to its properties (e.g., memory consumption), or properties of its sub-components (recursively), in a given context. The modes are composed from the variants with the highest utility in a given context (when context changes). In Genie [BGF+08], coarse-grained architecture modes, supplemented with explicit mode transition diagrams, are traced to low-level component configurations via orthogonal variability models [PBVDL05].

2.2.3 Dependable Unbounded (Self-) Adaptation

A significant body of work focuses on modeling architecture (self-) adaptation without bounding the potential adaptation space while still guaranteeing structural correctness. This is especially viable in domains that do not enable pre-designing all the architecture variants but still require dependability and prevention of architecture erosion.

2.2.3.1 Architecture (Self-) Adaptation Models at Runtime

The prevalent approaches to dependable unbounded (self-) adaptation [BJC05, DFB+12, GCH+04, HI10, KM07, KM09, MBNJ09, TGEM10] are based on modeling the adaptation at runtime and analyzing it together with constraints on the permissible architecture configurations so as to provide validation prior to the actual reconfiguration. Hence, these approaches heavily rely on models@runtime [BBF09, MBJ+09]. Note that they also typically involve formal methods, some of which are discussed in more detail in Section 2.3.1.

As to examples, the architecture models in Plastik [BJC05] (based on Acme [GMW00] and OpenCOM [CBG+08]) are equipped with formally captured invariants expressing consistency constraints on architecture (self-) adaptation. At runtime, every adaptation action (either pre-designed at the level of architecture, or ad-hoc) is first checked that it does not violate any architecture invariant.

Rainbow [GCH+04] presents a different view on unbounded (self-) adaptation. In addition to using architecture models to validate the effects of adaptation actions at runtime, a reflective middleware infrastructure is employed to update the architecture model by monitoring the executing system, while an adaptation layer analyzes the model against pre-defined architecture constraints. If any of them is violated, the adaptation layer employs the associated adaptation tactics.

The key advantage of these techniques, especially the ones that support automated adaptation planning – discussed further in Section 2.2.3.3, is that they allow for reasoning about the architecture (self-) adaptation at runtime. This is especially beneficial for architecting self-managed and autonomic systems, as elaborated in Section 2.2.3.4.

2.2.3.2 Architecture (Self-) Adaptation Models at Design Time

Another family of approaches to dependable unbounded (self-) adaptation is focusing on modeling the (self-) adaptation in such a way that correctness of all the potential architecture (re)configurations can be guaranteed at design time.

For example, FracToy [TMS10] (based on Fractal [BCL+06]) formally models architecture configurations, as well as architecture constraints and (self-) adaptation tactics, at design time so as to verify that the tactics will maintain the constraints at runtime. A similar approach is presented in FracL [SA09]; the difference to FracToy is that FracL models the adaptation tactics down to the level of the Fractal's control API. Conversely, in [GMK02] dependable (self-) adaptation tactics are derived at design time from a formal model of the desired architectural styles, represented via constraints on component composition in Darwin [MDEK95]. (The formalisms used in [GMK02], [SA09], and [TMS10] are described in Section 2.3.1.)

The approach of [BJM+11] employs formalized adaptation actions that are tailored to hierarchical component-based architectures (e.g., instantiation of a sub-component, creation of a component binding). These adaptation actions help to describe at design time when and how the architecture evolves at runtime in a way that facilitates formal dependability analysis and "correct-by-construction" code generation. The adaptation actions stem from well-defined reconfiguration patterns that are based on the authors' experience with non-trivial case studies [HP06].

Another similar approach is proposed in [EHH+13], where all the allowed architecture adaptation actions (determining when and how the runtime architecture evolves) are formally specified at design time. Dependability is then achieved by checking that these actions yield only architecture configurations corresponding to the verified component interaction patterns. In a similar vein, [HB13] proposes modeling (self-) adaptation in hierarchical real-time component-based architectures via a design-time formalization of the adaptation actions and their propagation along the component hierarchy. Each adaptation action is rigorously specified and annotated with real-time properties, thus governing both structural and real-time predictability. Correctness of such specification is then verified prior to deployment. (The formal aspects of [EHH+13] and [HB13] are discussed in Section 2.3.2).

By restricting (self-) adaptation to predefined actions/tactics, these approaches ensure predictability and dependability of all the potential architecture configurations at design time. This is imperative in safety-critical systems, including (some) RDS.

2.2.3.3 Automated Adaptation Planning

Some approaches that are based on modeling and analyzing architecture adaptation at runtime (Section 2.2.3.1), such as [BJC05], rely on explicit specification of low-level adaptation actions (tactics). This hinders the high-level design view on (self-) adaptation, binds the tactics to a given execution platform, and impedes determining the interactions between the tactics. Nevertheless, when maintaining an explicit architecture model together with constraints on the permissible architecture configurations at runtime, it is possible not only to validate the planned architecture adaptation tactics, but also to automate the planning itself. In this context, architecture adaptation planning determines the necessary reconfiguration steps to drive the adaptation from the current architecture to a given target architecture. In the following, we overview a few examples.

For instance, the approach of Plastik [BJC05] has been extended with support for automated architecture adaptation planning at runtime in [MBB+12] (further discussed in Section 2.3.1). A similar approach to adaptation planning is presented in [TGEM10].

In [MBNJ09], adopting an incremental approach to adaptation planning based on aspect-oriented programming [KLM+97], the adaptation tactics are represented via high-level aspect models that are woven into the explicit architecture model at runtime. The resulting architecture model is then validated on the fly (allowing simple rollback if needed). The actual adaptation is driven by low-level reconfiguration scripts that are automatically generated based on the differences between the initial model and the newly woven one. In addition to portability and appropriate abstraction level, using aspect models to represent adaptation tactics and automatically generate low-level reconfiguration scripts allows analyzing the interactions between the adaptation tactics.

The approach of [MBNJ09] is further elaborated in the context of the Kevoree component system [DFB+12][15], which focuses on runtime architecture models and adaptation planning for distributed, heterogeneous systems. Specifically, the runtime infrastructure of Kevoree relies on model comparison between two architecture models to generate an efficient reconfiguration script that adapts the architecture. Advantageously, this approach can also be effectively extended to other domains, such as cyber-physical systems [FMF+12].

Another example of adaptation planning is presented in [HI10], which (similarly to [GMK02] and FracToy [TMS10] mentioned in Section 2.2.3.2) relies on formal models of both the architecture structure and the adaptation actions. This approach employs formal analysis of the models at runtime (vs. design time) to enable online validation of architecture constraints (e.g., architectural style) and to perform the actual adaptation planning in the context of a generic component model based on OSGi [21]. For brevity, the employed formal method is discussed separately, together with other similar techniques, in Section 2.3.1.

2.2.3.4 Autonomic and Self-Managed Adaptation

A specific, partially crosscutting approach to architecture (self-) adaptation that has been given progressively more attention [BBF09, CDLG+09, DLGM+13, GCH+04, KM07, KM09, MPS08, PMC+12] pertains to the concerns of autonomicity and self-management [HM08, KC03]. In principle, the objective of autonomic and self-managed software architectures is to employ automated techniques for preparing architecture (self-) adaptation tactics in order to minimize the necessity of explicit architecture management while retaining compliance with overall architecture design and system goals in face of changing system requirements and environment. This is achieved by employing explicit autonomic feedback loops at the level of architecture models [BSG+09, MPS08].

The techniques that enable modeling and analyzing architecture adaptation at runtime (Section 2.2.3.1) are especially fitting for this purpose. Specifically, they bring the possibility to reason about architecture adaptation within the explicit autonomic feedback loop. This particularly pertains to techniques supporting automated architecture adaptation planning (Section 2.2.3.3), since the adaptation plans, and their rigorous models and analysis, play a critical role in achieving reliability in autonomic and self-managed systems. To react to a changing application context in an autonomic manner, these techniques need to be complemented by adequate monitoring support in terms of architecture reflection [ADLMW09], such as the OSGi [21] reflective API.

To reduce the design complexity of autonomic adaptation, it is beneficial to employ a more elaborate organization of feedback loops.

For example, layered self-managing architecture organization may help separate different adaptation domains within a software architecture [KM07, KM09]. In this case, a goal management layer is responsible for deriving adaptation plans from high-level goals, such plans are managed and transformed into adaptation actions in the change management layer, and, finally, the component control executes these actions and reports on the status of the newly derived architecture.

Hierarchical organization, especially in combination with model-based approaches to architecture adaptation [MBJ+09], can be efficiently employed to achieve self-managed evolution of both software architectures and their adaptation facilities [PMC+12]. Specifically, this can be done by treating the architecture adaptation loop as another autonomic and self-managed system equipped with its own adaptation feedback loop. The goal of this second loop is meta-adaptation, i.e., adaptation of the original feedback loop as a reaction to, e.g., new requirements or new adaptation dimensions.

2.2.4 Inherently Dynamic Architectures

In the previous subsections, we have been focusing on explicit architecture (self-) adaptation via reconfiguration. A distinct view on (self-) adaptation is embodied by approaches where the software architecture itself is inherently dynamic, i.e., it adapts (or is being adapted) without explicit changes to the architecture design (e.g., by reconfiguration). Naturally, these approaches rely heavily on an execution environment, which manages all the actual architecture (self-) adaptation [EHL07]. Since an inherently dynamic architecture gets fully concretized as late as at runtime, it is often dubbed as an *implicit architecture* (i.e., it only implicitly describes the actual runtime architecture). This is advantageously utilized in pervasive distributed systems featuring high mobility or dynamic availability of components in an open environment, where adaptation via explicit reconfiguration may not be plausible due to the openness or large scale.

The concept of inherently dynamic architectures in fact stands behind most of the approaches focusing on service-based systems. For example, each service binding in a service-based architecture (Section 2.1.3) is based on dynamic service discovery managed by a service broker. This way, although static in its design, the service-based architecture is adapted at runtime (by the service platform) according to the dynamic availability of the corresponding services [EHL07].

Similarly, architectures of multi-agent systems (Section 2.1.4) are inherently dynamic in a sense that the static definition of groups and roles forming the architecture design is interpreted by agents at runtime so as to achieve dynamic self-organization of the architecture [FGM04].

Finally, inherent dynamism is the key concept of architectures based on service-component ensembles (Section 2.1.5). In particular, due to the attribute-based communication of components in an ensemble, every component interaction is interpreted dynamically based on the current runtime context of the (potentially) interacting components [DNLPT14].

Note that inherently dynamic architectures can be advantageously combined with architecture (self-) adaptation via reconfiguration (e.g., Section 2.2.3) so as to achieve a higher degree of dynamism. This approach has been for example employed for dynamic reconfiguration of service-component architectures [IFMW08, TSP+04]. Such a combination is especially important when designing self-managed and autonomic systems (Section 2.2.3.4) on top of inherently dynamic architectures.

2.2.5 Lessons Learned

Building on the discussion of architecture abstractions suitable for RDS in Section 2.1, the previous sections have further detailed the fundamental approaches to architecture (self-) adaptation. The approaches offer different tradeoffs between the scope of architecture dynamism (important w.r.t. the challenges C1-C3) and predictability, analyzability, and dependability (relevant to the challenge C4), and thus provide different utility for RDS. Note that although appropriate (self-) adaptation mechanisms are essential for achieving the goals **G1** and **G3**, they are equally important w.r.t. the goal **G2**.

Ad-hoc architecture (self-) adaptation, discussed in Section 2.2.1, is on one hand extremely flexible and open-ended. On the other hand, since it is not explicitly modeled at the architecture level, it makes it hard or even impossible to ensure dependability, which may result in architecture erosion. This makes ad-hoc adaptation inapplicable for RDS.

The other extreme is represented by bounded architecture (self-) adaptation, discussed in Section 2.2.2. It promotes predictability and dependability by limiting the architecture dynamism to the pre-designed architecture configurations. Although this makes architecture (self-) adaptation much more manageable at both design time and runtime, it is too restrictive w.r.t. the extremely dynamic and open environments of RDS to be generally applicable for their distributed architectures. Nevertheless, its wide spread in the area of embedded and real-time systems makes bounded (self-) adaptation a feasible alternative for adaptation of local (i.e., non-distributed) RDS sub-systems that are responsible for interaction with the physical environment.

The approaches to unbounded (but constrained) architecture (self-) adaptation, outlined in Section 2.2.3, offer a balanced tradeoff between flexibility and predictability. Especially appealing is the concept of modeling and analyzing architecture adaptation at runtime, since it enables automated planning of complex reconfigurations, as well as fully autonomic and self-managed adaptation in face of changing system requirements and environment. In this context, employing explicit autonomic feedback loops at the level of architecture appears to be one of the key ingredients. Nevertheless, since this type of (self-) adaptation involves various reasoning and analysis techniques, it may not be appropriate for addressing very frequent architecture adaptation that is caused by changes of component reachability or context, regularly manifested in RDS. In fact, such cases mainly require only small adjustments of the architecture in terms of component re-connection rather than a full architecture reconfiguration. Moreover, unbounded (self-) adaptation is based on modeling the overall system architecture at runtime, and thus it is not amenable to efficient decentralized execution required for RDS. On the other hand, it may still be a suitable alternative for coarse-grained architecture (self-) adaptation (e.g., meta-adaptation – adaptation of adaptation mechanisms), where a more costly or centralized approach is tolerable and where it can utilized to its full potential. In this context, maintaining an architecture model at runtime and architecture monitoring are the key prerequisites.

The closest match to the needs of RDS are actually the inherently dynamic architectures discussed in Section 2.2.4, which are at design time specified only implicitly and are fully concretized at runtime by the execution environment. Although this approach is not flexible enough to express arbitrary architecture reconfigurations (i.e., the overall design-time blueprint of the architecture is fixed), it is suitable for addressing the recurrent dynamism of component bindings w.r.t. dynamic component availability and changes of context (e.g., due to high mobility or component/communication failures). It also facilitates open-ended architecture design, while retaining predictability. However, to fully exploit the potential of inherent architecture dynamism in face of recurrent architecture adaptation and unstable communication links, it is necessary to provide appropriate communication mechanisms that would mitigate the impact of frequent architecture changes on the individual components, which is one of the major weaknesses of the state-of-the-art approaches in this area. To this end, a stateless, data-flow-based communication is an appealing candidate. Further, the significant role of the execution environment in establishing the actual architecture at runtime makes it problematic to ensure dependability without a proper, rigorously-specified operational semantics of the execution environment (as elaborated further in Section 2.3).

2.3 Formal Methods in (Dynamic) Software Architectures

Formal methods, formal specification and analysis in particular, play an important role in ensuring dependability of (dynamic) software architectures, which is critical in RDS. Since formal specification and analysis are not the primary objectives of this thesis, this section overviews only certain influential examples. Specifically, we focus primarily on the approaches that are related to Sections 2.1 and 2.2.

To this end, we first elaborate on approaches to structural specification and analysis of (dynamic) software architectures (Section 2.3.1), which are concerned purely with architecture (re)configurations without considering the behavior governed by the architectures. This also includes techniques for dependable automated construction of architecture configurations. Then, we discuss approaches to behavior specification and analysis (Section 2.3.2), which are concerned with the overall system behavior and properties governed by the (dynamic) architectures. Specifically, we discuss three major directions: (i) specification and analysis of a formal model of an architecture, (ii) specification and analysis of an actual implementation following the architecture (a.k.a. code verification), and (iii) specification and analysis of system requirements determining the architecture. Finally, we conclude this section with a discussion of how the presented approaches contribute to ensuring dependability in the dynamic architectures of RDS (Section 2.3.3).

Note that the domains of structural and behavior analysis of software architectures are not necessarily exclusive. An interesting illustration of how formal methods can be employed in the context of both is presented in [GS07].

2.3.1 Structural Specification and Analysis

When analyzing (dynamic) software architectures, a significant body of work primarily focuses on the structural properties. In this context, the goal is to formally model and analyze the (static) structure of software architecture configurations, potentially in combination with the (dynamic) reconfiguration actions that drive architecture adaptation, while abstracting the actual system behavior.

Logic-based constraint-solving techniques are especially suitable for this purpose. Not only do they allow for convenient structural modeling of architecture configurations, but they can be also advantageously employed for their automated construction in a way that guarantees dependability (i.e., by interpreting a solution of a constraint-satisfaction problem that properly encodes all admissible configurations as a concrete configuration).

In this context, several approaches focus on SAT (propositional satisfiability) [BHvMW09]. In [LBP+08, MBDC+06] the idea is to employ a SAT solver to resolve software dependencies; this approach was used in several contemporary software tools such as Equinox p2 [9] (which is an implementation of OSGi [21]) and Maven [3]. Equally, the approach of [TBKC07] is based on formalizing the notion of a feature model by introducing a specific feature algebra. Using such formalization, a SAT solver is employed for verification of safe feature-model composition in a product line.

A very popular logic-based constraint-solving technique suitable for formal structural architecture specification and analysis is based on employing first-order finite structures (i.e., first-order logic predicates upon finite sets and relations).

In this context, Alloy [Jac02, Jac12] stands as one of the most popular tools for structural specification and analysis of software architectures, as well as their automated construction [DR14, GMK02, HI10, JS00, KG10, MS+08, SA09, TMS10]. Due to the large number of applications, we discuss Alloy and the related approaches separately in Section 2.3.1.1.

Another technique for structural analysis and automated construction of architecture configurations based on first-order finite structures is logic programming. For example, the approach of [Her10] uses first-order finite structures to describe component-based architectures based on UML component diagrams. The formal description includes component ports and connectors, as well as individual methods of component interfaces, their parameter types, and basic characteristics of their bodies (instances of what classes/component a particular method can create/destroy, etc.). The same formalism is employed to describe architecture rules, which act as architectural patterns, reference architecture constraints, guidelines and policies, etc. A Prolog-like logic-programming framework is then used for design-time verification of a given architecture against a given set of architecture rules. Another example is [BP04], where logic programming is used in order to formalize hierarchical composition of software connectors from reusable elements in a way that reflects both functional (e.g., communication style, interface signatures) and extra-functional (e.g., transactions, monitoring) concerns. Prolog is then employed to construct a particular connector architecture.

Recently, approaches to structural architecture analysis and automated construction based on automated planning [GNT04] have gained attention. This includes automated adaptation planning in self-adaptive component-based software architectures [MBB+12, TGEM10] (introduced in Section 2.2.3.3), composition of services in service-oriented architectures [SW04], and automated architecture evolution [BPG13]. Analogously to the previous techniques, these approaches describe software architecture via logic-based predicates expressing relations among various finite structures (e.g., a predicate expressing that a given component and a given connector are connected to each other). Similarly, the constraints to be maintained during adaptation (e.g., component model, architectural style) are expressed via first-order logic invariants restricting admissible reconfigurations (with the option to also constraint whole reconfiguration paths). Advantageously, the approaches based on automated planning enable also reflecting concerns such as timing constraints and optimization w.r.t. to a custom metric (e.g., minimizing the cost of reconfiguration).

2.3.1.1 Analysis of (Dynamic) Software Architectures with Alloy

Alloy [Jac02, Jac12][1] is a formal modeling language that is based on a first-order predicate logic with operators from the set theory (e.g., intersection, cartesian product), relational algebra (e.g., relational join, transitive closure), and basic arithmetic (e.g., integer operations, set cardinality). The language is based on the notions of signature and relation. A signature is a set of abstract elements; relations are defined upon such sets. Alloy allows constraining the relations by facts (first-order logic formulas). A fact can employ named predicates and function symbols. In general, an Alloy specification, which consists of definitions of signatures, relations, and facts, represents a first-order logic theory (also called Alloy model). Alloy Analyzer – the associated solver – can either find an instance of an Alloy model (i.e., assign elements to signatures and establish relations among the elements in a way that satisfies the constraints), or check a model (i.e., all its possible instances) against a given property (expressed as a constraint). Alloy Analyzer converts each Alloy model to a SAT formula [BHvMW09]; using an underlying general-purpose SAT solver, it solves the formula and then interprets the result as an Alloy model instance. Consequently, Alloy Analyzer requires the domains of signatures and relations to be explicitly bounded (due to the mapping to SAT). The technique of translating a model analysis to SAT is sometimes called bounded model checking [BCC+09].

It is due to its syntax, compatible with the object-oriented paradigm, and features of Alloy Analyzer that Alloy has been extensively used in the domain of software architectures for the purpose of both architecture analysis (via property checking) and automated composition/synthesis (via instance finding).

The below-discussed approaches, some of which were already introduced in Section 2.2.3, typically model architectures in the following way: Structural elements of an architecture (e.g., components, interfaces) are represented via signatures and their organization via relations (e.g., a relation between a component and its interfaces). An adaptation action (tactic) is then represented as a predicate describing pre- and post-conditions of the action in terms of constraints over the source and target architecture configurations. Consequently, the architecture adaptation process is modelled via a sequence of architecture configurations, where each two consecutive ones satisfy the pre- and post- conditions of an adaptation action.

In [GMK02] the authors employ Alloy for specification and analysis of structural properties of self-adaptive architectures in Darwin [MDEK95, MK96]. In this context, Alloy Analyzer is employed for analysis of a specified architecture configuration (encoded into an Alloy model) w.r.t. predefined architectural constraints (encoded into Alloy facts). The Analyzer is also used for determining suitable adaptation tactics (by finding an instance of the corresponding Alloy model).

Another Alloy-based approach to verification of (self-) adaptation is FracToy [TMS10]. Similarly to [GMK02], it is based on a formalization of a concrete self-adaptive architecture and the corresponding adaptation tactics in terms of an Alloy model so as to analyze the effect of the adaptation tactics on the architecture w.r.t. predefined architecture constraints.

In a similar way, [LLC10] employs Alloy for specification and analysis of architecture integrity constraints so as to validate architecture reconfiguration at runtime (in terms of Section 2.2.3.1).

The approach presented in [HI10] employs Alloy for specifications and analysis of architecture adaptation in two ways: (a) verification of adaptation tactics and (b) adaptation planning. In alignment with the previous approaches, the former focuses on soundness of the adaptation tactics and preservation of the properties specific to a particular architectural style when these tactics are applied to a particular architecture configuration (by checking against Alloy facts). The latter involves finding a fitting sequence of adaptation actions from the current architecture configuration to a given configuration while preserving a particular architectural style in all intermediate states (by finding instances of the corresponding signatures and relations).

Although not directly based on Alloy, FracL [SA09] also employs first-order finite structures to formalize static architecture configurations and primitive adaptation actions, while focusing on design-time adaptation analysis. Alloy is used as one of the analysis options (the other being the Focal [DHDG06] proof framework) for verification of the adaptation actions when applied to a concrete architecture configuration. FracL also allows for specifying procedural reconfiguration scripts based on the primitive adaptation actions. Their correctness, however, needs to be shown via formal proofs based Hoare logic using theorem-proving tools.

As to other applications, Alloy is employed in [KG10] for formal specification and design-time analysis of various architectural styles, with the focus on verifying consistency, composability, and refinement of the architectural styles. The Alloy model of an architectural style is obtained programmatically from its Acme [GMW00] specification.

Focusing on a higher level of abstraction, [JS00] introduces a formal Alloy specification of valid component compositions in the COM component model [Box98]. Similarly, [MS+08] presents a fully-fledged Alloy formalization of the Fractal component model [BCL+06]. Both of these approaches focus primarily on analyzing the properties of the component models themselves (rather than properties of particular architectures based on the component models, as in the previous cases).

2.3.2 Behavior Specification and Analysis

2.3.2.1 Formal Specification and Analysis of Architecture Models

A large body of work focuses primarily on formal specification and analysis of overall system behavior and properties governed by an architecture, especially in combination with architecture dynamism. For this purpose, operational semantics of both the business logic and the architecture (e.g., component communication protocols, execution model) needs to be captured formally.

In the context of this thesis, formalization of the architecture operational semantics is of primary interest, as it provides an important baseline for formal analysis of architecture dynamism. Moreover, it provides a rigorous, in-depth description of the key architecture abstractions, their runtime behavior, and properties, which plays an important role when implementing or employing the abstractions. For example, the Darwin component model [MDEK95], as well as its dynamic fragment [MK96], is formalized in terms of π-calculus [MPW92] so as to capture a precise semantics of its concepts. Another example is Wright [AG97], semantics of which is based on the CSP [Hoa85] process algebra. Similarly, [HKR09] presents a precise formalization of the GCM component model [BCD+09, BHR14], providing a reliable basis for potential implementation and formal analysis; this formalization is then employed in [HK10] so as to verify correctness of the GCM semantics w.r.t. asynchronous component execution using the theorem prover Isabelle/HOL [NWP02].

As to formal analysis, a significant body of work focuses on analysis of formal architecture specifications via model checking [CGP99], as surveyed, e.g., in [ZML10]. Model checking is an exhaustive automatic verification technique enabling verification of conformance of a given model (e.g., architecture operational semantics) to expected properties (e.g., safety, liveness) that is based on exploration of the state space determined by the model. The properties are usually expressed in a dedicated property language (e.g., linear temporal logic – LTL [VW86]). Note that all the model-checking approaches typically need to deal with the issue of state space explosion. Following the idea of architecture decomposition into separate components, when model checking software architectures, this issue is often addressed via compositional verification that is based on the assume-guarantee reasoning [CAC08, GPC04, Pnu85, PP10]. Since there is a large body of work in this area, we briefly list only representative examples.

For instance, in [MKG99] the finite state process (FSP) algebra was used to capture operational semantics of Darwin-based software architectures [MDEK95] in terms of labeled transition systems (LTS), in order to enable formal analysis (using the LTSA model checker [MK99]) of safety and liveness properties (expressed in the corresponding FLTL temporal logic).

Especially well known in this area is the behavior, interaction, priority (BIP) formal component framework [BBB+11, BBS06]. It targets formal modeling and verification of explicit component-based architectures that consist of hierarchically structured components featuring formally defined interfaces and (real-time) behaviors (described via annotated finite-state automata). The explicit BIP connectors [BS08] determine the possible interactions to synchronize the concurrently executing components. The priorities then represent scheduling policies for these interactions. BIP is accompanied with a large variety of tools, including the D-Finder [BBNS09] model checker for compositional verification of deadlock-freedom and other safety properties, as well as tools for a "correct-by-construction" generation of code from a BIP model. BIP has been also extended with support for (bounded) architecture adaptation [BJMS12] and statistical model checking [BBB+12] (employing stochastic abstractions to allow verification of large heterogeneous systems).

Focusing on dynamic and mobile architectures, π-ADL [Oqu04] is a formally grounded architecture description language based on higher-order typed π-calculus [San93] process algebra. Similar to BIP, it covers both structural and behavioral viewpoint of component-based software architectures. The structural viewpoint offers hierarchical composition of components with explicit component types and connectors. The behavioral viewpoint enables expressing behaviors of components (acting as autonomous agents), port protocols, as well as (bounded) architecture adaptation and component mobility. It is also equipped with formal analysis techniques based on model checking via CADP [GLMS11] and theorem proving via XSB Prolog [23]. The analyzable properties include general temporal properties, such as safety and liveness, as well as compatibility or equivalence between different components and connectors. The properties are expressed in the dedicated property language π-AAL [MO06], which combines predicate logic with temporal logic,

The MECHATRONIC UML approach [BGT05], targeting formal architectural modeling of distributed safety-critical real-time component-based systems, also reflects both structural viewpoint (through refined UML component diagrams) and behavioral viewpoint (through real-time extension of UML state-chart diagrams). Advantageously, it allows for specification of hybrid component behaviors (i.e., featuring both discrete and continuous activities), interface contracts, and component coordination patterns. The analysis is enabled through translation into timed automata and subsequent analysis with UPPAAL [LPY97]. Thus, the analyzable properties, specified via a variant of TCTL [ACD93], include safety properties and timed liveness properties of component behaviors and coordination patterns. This approach has been extended in [BBG+06, EHH+13] with support for formal analysis of (constrained) architecture (self-) adaptation. This extension is based on formalizing allowed architecture reconfigurations (via component story diagrams) and checking that they yield only architectures corresponding to the predefined interaction patterns. The safety and liveness properties can be then verified by individually analyzing the interaction patterns (formalized via real-time state-charts). Similarly, [HB13] presents an extension focusing on formal modeling and timed model checking of reconfigurations in hierarchical component-based architectures. Specifically, it focuses on declarative formal specification of real-time architecture reconfiguration and its propagation along the component hierarchy (based on propagation of reconfiguration messages that are associated with declarative reconfiguration rules expressed via component story diagrams). This declarative specification is then automatically operationalized in terms of real-time state-chart diagrams and, similar to [BGT05], verified w.r.t. deadlock freedom and timed safety properties (i.e., through translation into timed automata and subsequent analysis with UPPAAL [LPY97]).

There is a plethora of other similar approaches to formal architecture specification and analysis, from which we will name just a few. For example, [VSC+09] presents a formal operational semantics of the ProCom component model [SVB+08] so as to clarify the concepts and enable formal analysis. Further, component interaction in SOFA 2 [BHP06] can be captured in terms of formal behavior protocols [PSPK13]. The protocols enable compositional verification of correctness based on the absence of communication errors and verification of hierarchical component compositions based on a formal notion of refinement. A general approach to formal modeling and analysis of adaptive architectures, focusing on separation of adaptation behavior and functional behavior, is presented in [ZC06]. In addition, there exists a number of well-established modeling languages and tools, such as AADL [FGH06] and the Ocarina tool suite [HZPK08], focusing primarily on timing and dependability analysis. Similarly, the CHESS [7] methodology and toolset are mainly aiming at timing analysis, failure propagation, and dependability analysis. Targeting embedded systems, the approach of Othello/OCRA [CDT13, CT12] allows for checking refinement of contracts expressed in a variant of linear-time temporal logic interpreted over hybrid traces (i.e., traces that contain both discrete events and continuous-time state evolution). Assuming a different perspective – the software performance perspective – the Palladio component model [BKR09] focuses on formal specification, analysis, and prediction of performance-related extra-functional properties of component-based architectures. Note that, instead of model checking, the analysis in Palladio primarily relies on simulation and techniques specific to the domain of performance prediction.

There are also many approaches targeting specifically service-oriented architectures. This includes techniques for formal specification and model checking of (web) service compositions [FUMK03] and choreographies [BWR09], as well as formal specification and verification of service-oriented architectures in general [BBC+06, FL10].

The idea of providing formal foundation for the system design has been also applied in the area of (multi) agent systems, employing, e.g., the concepts of declarative (logic) programming to capture the dynamism in adaptive systems [LN11]. Similarly, there are also approaches presenting formally grounded models for rigorous specification of (multi) agent systems and their subsequent verification via model checking [BFVW06, LQR09, WFHP02].

As to the design of (self-) adaptive systems in general (not necessarily focusing on the architecture), there is a large body of work focusing on formal modeling methods that are based on process-algebraic coordination languages [CG98, DNFP98, DNLPT14], including methods for modeling autonomic behavior based on the notion of feedback loop [GLPT12]. These approaches also aim at addressing the challenges connected to distribution, locality, and mobility [BLP01, CG98, CL10, DNL04]. A comprehensive overview of this domain is, however, outside the scope of this thesis.

Another large body of work focuses on quantitative verification [Kwi07] of software architectures. For example, in [BBCO12] quantitative verification was employed for analysis of component-based systems modelled as Markov decision processes so as to verify their reliability properties (e.g., probability of failure, failure frequency, or deadlock). Quantitative verification can also be advantageously employed in the domain of self-adaptive systems [CGKM12] in order to ensure correct adaptation at runtime. Nevertheless, detailed discussion of this domain is also beyond the scope of this thesis.

2.3.2.2 Code Verification

A very important approach in the context of behavior analysis of (dynamic) software architectures is code verification, which targets software implementation. Specifically, it can be advantageously employed to analyze behavior aspects of software architecture, especially w.r.t. low-level implementation concerns. Code verification is also often combined with model-based formal analysis, e.g., [ASCN03, BHH+06, JPK12, PPK06].

In this context, one of the main approaches is code model checking, which is based on exploration of the state space defined by the code. The explicit-state code model checking techniques represent the model states so that their structure and transitions directly correspond to the in-memory state of the analyzed code. As a result, they allow addressing low-level concerns, such as null pointer and invalid memory accesses, assertion violations, race conditions, and deadlocks. As to examples, Java PathFinder [VHB+03][16] is an explicit-state code model checker for Java, based on a modified Java virtual machine specialized for model checking of Java bytecode. Similarly, GMC [11] focuses on explicit-state model checking of code in C and C++. Note that architecture environment, including, e.g., the employed component platform and middleware, needs to be reflected in the verified code, i.e., modeled or directly included.

In addition to explicit-state code model checking, many other techniques for code verification exist. Examples are abstraction-based code model checking [BMMR01], trading-off scalability for variable precision, or deductive code verification [BCC+05], based on transforming the code and the verified property into a formal model and analyzing it via a theorem prover. A comprehensive overview of code-model-checking techniques can be found, e.g., in [Ser10].

2.3.2.3 Formal Specification and Analysis of Requirements

Similar to code verification (Section 2.3.2.2), formal specification and analysis of requirements is another area that is not directly bound to software architectures, but is essential for ensuring correct architecture design. In particular, formal requirements specification and analysis may help designing dependable software architectures [BPG+04, TPYZ09, VL03, YLL+08] and their intended high-level behaviors [PMM+07]. Since this area is relevant to the presented contributions but its comprehensive discussion of is beyond the scope of this thesis, we describe only a few influential examples.

The main body of work in this area is represented by the Goal-Oriented Requirements Engineering (GORE) [RBAF10, VL01]. In this context, KAOS [PMM+07, VLL04] is a prominent approach that is equipped with a rich set of formal analysis techniques. The main idea of KAOS is based on goal refinement. High-level goals, capturing global, strategic objectives, are systematically refined into low-level goals that capture local, technical objectives, realizable by actual software or environment agents. (Thus, KAOS explicitly distinguishes software requirements from expectations and assumptions.) Semantics of a goal can be formally captured in a real-time linear temporal logic, as elaborated for example in [VLDL98, VLL00]. This formalization helps ensuring correctness of goal refinement [DVL96] and goal operationalization [LVL02], which is in KAOS primarily achieved by application of refinement patterns, correctness of which is formally proven (once and for all) via a theorem prover. The formalization also enables automatic verification of consistency and completeness of the goal models via model checking [PMM+07]. Moreover, formal analysis can be also employed for conflict management [VLDL98] and obstacle (hazard, threat) analysis [VLL00]. Given an operationalized formal goal model, it is possible to derive a "correct-by-construction" architecture model [VL03] (similar to Tropos [BPG+04], discussed in Section 2.1.4). In this context, several approaches focus specifically on (self-) adaptive systems [BCGZ06, TPYZ09, YLL+08].

In a similar vein, [FPMT01] presents an approach to formal specification and analysis of requirements based on Tropos [BPG+04] that covers formal verification of invariant properties and assertions via model checking.

State of the affairs (SOTA) [ABZ12] model represents an approach specifically targeting high-level requirements in the domain of self-adaptive systems. The key idea of SOTA is to abstract the behavior of a system in terms of a trajectory through a multi-dimensional space – SOTA space – that reflects all the possible contexts of the system. The requirements of a system in SOTA are captured in terms of goals. A goal is an area in the SOTA space that a system should eventually reach, and it can be characterized by its pre-condition, post-condition, and utilities (i.e., constraints on the path to the goal). Compared to the informal approach of Tropos, SOTA directly provides the means for formal modeling of early requirements and their operationalization. Consequently, it also enables verification of goal operationalization and detection of inconsistencies or implicit requirements via model checking [AZ12].

2.3.3 Lessons Learned

Focusing on the areas related to Sections 2.1 and 2.2, Section 2.3 has overviewed the principal approaches to formal specification and analysis of software architectures with the aim of providing a foundation for addressing the challenge of dependability (C4) throughout the research goals **G1**-**G3** (which themselves put a strong emphasis on the formal and semantic aspects).

The part devoted to structural specification and analysis, i.e., Section 2.3.1, has shown that formal methods can be conveniently employed for ensuring dependability of dynamic architectures even when abstracting the actual system behavior. Especially appealing is the idea of employing formal analysis, constraint solving in particular, not only to verify structural correctness of architecture (re)configurations (primarily considered by the research goal **G3**), but also to automate their construction (which is part of the research goal **G2**). Due to its wide spread and popularity in this area, the Alloy modelling language appears to be a suitable tool for this purpose.

In the remaining part, i.e., Section 2.3.2, it has been shown that behavior specification and analysis is a widely adopted technique for ensuring dependability of (dynamic) software architectures. A key ingredient is in this case a formalized operational semantics of the architectures (which is also part of the research goals **G2** and **G3**). Not only does it enable formal analysis of the overall behavior governed by an architecture, but it also provides a rigorous description of the key architecture abstractions, which is important for their correct adoption and implementation. However, due to the need of tailored architecture abstractions, there is no readily-applicable solution for RDS. As to formal analysis (which is an important part of the research goal **G3**), model checking represents the dominant approach employed throughout various domains focusing on (dynamic) software architectures. Nevertheless, it still faces several unresolved issues w.r.t.

the challenges posed by RDS, which need to be resolved in order to ensure dependability of dynamic RDS architectures. In particular, it is disputable, whether the extremely open and dynamic architectures of RDS are feasible in terms of scalability and expressive power of the state-of-the-art model-checking techniques. Another useful analysis approach is code verification, which complements model checking by ensuring dependability of software architectures at the implementation level. In this context, explicit state code model checking is of particular interest, as it enables addressing low-level concerns stemming from distributed, decentralized execution of RDS, such as race conditions. Note, however, that appropriate representation of the architecture environment in the verified code (encapsulating the semantics of the corresponding architecture abstractions and execution platform) is the key for ensuring feasibility of code verification in RDS. Finally, approaches to formal specification and analysis based on model checking have been applied also in the domain of requirements engineering and agent-based system design so as to provide dependability guarantees during the early architecture design. It appears that these approaches can be advantageously exploited in the context of RDS-specific architecture abstractions, thus bringing the formal aspects to the RDS architecture design process (as required by the research goal **G2**).

Chapter 3

Collection of Papers

The main contributions of this thesis were published separately in various international journals and conference proceedings. This chapter includes both summaries and full versions of the selected relevant papers in the order presented in Section 1.5, as well as comments on the journals and conferences where the papers were presented.

The first paper included in this chapter – *Towards Dependable Emergent Ensembles of Components: The DEECo Component Model* [KBPK12] – portrays a vision of the DEECo component model, which introduces novel architecture abstractions that enable designing inherently dynamic software architectures for RDS, as further detailed in Section 3.1. It was presented at the Working IEEE/IFIP Conference on Software Architecture (WICSA) in 2012, ranked A in the CORE conference ranking [6]. The paper is cited in [BBCP13].

The second included paper – *DEECo: an Ensemble-Based Component System* [BGH+13] – develops the vision of [KBPK12] by fully elaborating the architecture abstractions of DEECo and supplementing them with an execution environment and a software development process, as further summarized in Section 3.2. In 2013, the paper was presented at the International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE), which is one of the Federated Events on Component-Based Software Engineering and Software Architecture (CompArch). The conference is ranked A in the CORE conference ranking [6]. The paper is cited in [HK14] and [BHR14].

Design of Ensemble-Based Component Systems by Invariant Refinement [KBP+13] is the third included paper. As described in Section 3.3, it presents the Invariant Refinement Method (IRM) – a goal-oriented design method targeting systems built upon the architecture abstractions of DEECo, which were introduced in the previous two papers. Similarly to [BGH+13], this paper was presented at the International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE) in 2013, ranked A in the CORE conference ranking [6], where it was awarded with the Distinguished Paper Award.

The fourth paper – *Automated resolution of connector architectures using constraint solving (ARCAS method)* [KBPH14] – describes a method for open-ended design and automated synthesis of software connectors that is particularly suitable for realization of the dynamic, emergent component bindings required in RDS, as further elaborated in Section 3.4. It was published in the Software and Systems Modeling journal in 2014, which on the date the paper was accepted had impact factor 1,250 and acceptance ratio of 14%. The paper is cited in [DR14].

In the fifth paper – *Towards Verification of Ensemble-Based Component Systems* [BBB+13] – we have presented a formalization of the general semantic model of DEECo and discussed the opportunities for verification of DEECo-based applications, as summarized in Section 3.5. The paper was presented in 2013 at the International Symposium on Formal Aspects of Component Software (FACS).

Finally, the last paper – *Adaptive Deployment in Ad-Hoc Systems Using Emergent Component Ensembles: Vision Paper* [BBHK13] – describes a vision of a realistic DEECo-based architecture for adaptive deployment in ad-hoc cloud systems, as overviewed in Section 3.6. The paper was presented at the ACM/SPEC International Conference on Performance Engineering (ICPE) in 2013.

3.1 Towards Dependable Emergent Ensembles of Components: The DEECo Component Model

Jaroslav Keznikl,
Tomáš Bureš,
František Plášil,
Michal Kit

In proceedings of the **Joint 10th Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture (WICSA/ECSA '12)**.

Published by IEEE CS,
pages 249-252,
ISBN 978-0-7695-4827-2,
August 2012.

The original version is available electronically from the publisher's site at http://dx.doi.org/10.1109/WICSA-ECSA.212.39.

Summary of the Paper

This paper, published as [KBPK12], is the baseline for the other contributions presented in this thesis. It tackles the challenge of designing large-scale distributed software architectures that are composed of autonomous components and employ continuous architecture self-adaptation so as to cope with their open, highly dynamic environments (i.e., challenges C1-C3 outlined in Section 1.3). In particular, the focus is on the excessive communication-link instability and architecture dynamism.

Attacking the research goal **G1**, the paper responds to this challenge by presenting a vision of novel architecture abstractions – summarized in the thereby introduced *DEECo component model* [8] (Dependable Emergent Ensembles of Components) – that overcome the design complexity of such systems via proper separation of concerns. The objective is to move the responsibility of dealing with excessive communication-link instability and architecture dynamism from the system/component designer to the component model (its execution environment in particular) in a way that reduces the design complexity and allows efficiently dealing with these challenges in the execution environment. By giving a clear description of the responsibilities of the execution environment, the paper also brings the first insights for tackling the research goal **G3**.

The basic idea of DEECo is to extract the concerns pertaining to adaptation of component bindings at runtime into a declarative, design-time description and to manage the adaptation in an automated way via the envisioned DEECo execution environment. In this respect, as illustrated in [SBK13], DEECo is inspired by the idea of service-component ensembles (SCEs) [HRW08], adopting at the conceptual level the fundamentals of the SCEL specification language [DNFLP13, DNLPT14]. Namely, it adopts the idea of attribute-based communication, which is employed to model a dynamic interaction of autonomous components within SCEs. To this end, DEECo brings two important contributions. First, SCEs are explicitly captured as first-class citizens at the level of software architecture, thus facilitating architecture design of SCEs. Second, the corresponding attribute-based communication is implicit, i.e., driven by the execution environment. Moreover, the attribute-based communication in DEECo takes the form of a stateless data flow among the components within SCEs. Thus, a component can be considered as an autonomous and self-aware entity, relying solely on its local state, which is modified in the background by the execution environment according to the attribute-based communication governed by the component's involvement in SCEs. This promotes architecture predictability, scalability, and resilience in face of unreliable communication links and/or rapid architecture changes.

Going into more detail, a component in DEECo is an autonomous unit of design, composition, and deployment, encapsulating its internal state – represented as a hierarchical data structure termed *knowledge* – and behavior – captured by a set of *processes*. A process is essentially a soft-real-time, cyclic task that, similar to a feedback loop, recurrently manipulates the knowledge of the component.

Component composition is in DEECo expressed implicitly via a dynamic involvement in ad-hoc groups of components called *ensembles*. As such, ensembles in DEECo are first-class concepts. In general, the involvement in an ensemble is determined by the ensemble's *membership condition* – a declaratively-expressed condition specifying which components are to participate in the ensemble, based on their attributes. (This also allows the components to affect their involvement in the ensemble by modifying the relevant attributes.) Communication among components is limited to implicit *knowledge exchange* – a stateless, best-effort data flow that is transferring part of the knowledge of one component to another within the same ensemble. In this context, an ensemble plays the role of a dynamic connector that enables DEECo architecture to emerge at runtime by dynamically connecting the components that meet the membership condition and performing the implicit knowledge exchange. Technically, each ensemble is instantiated by the execution environment according to a design-time ensemble prescription specifying the membership condition and knowledge exchange. Internally, an ensemble is structured to a unique coordinator and several member components. An ensemble prescription defines membership as a condition (in the paper referred to as membership function), under which two components form a coordinator-member pair, and knowledge exchange (in the paper referred to as mapping function) as a function mapping the coordinator's knowledge to the knowledge of a member (and vice versa). Note that the definition of membership and knowledge exchange is limited to coordinator-member pairs (rather than general tuples of one coordinator and multiple members) to decrease the runtime complexity of establishing ensembles. To facilitate reusability and separation of concerns, coordinator/member components are in an ensemble prescription referred via *interfaces*, rather than being specified directly. The interfaces provide a partial view on the components' knowledge and are associated with the components via structural subtyping (i.e., duck typing).

Concerning the computational model of DEECo, execution of both the component processes and ensemble knowledge exchange is driven in a cyclic, soft-real-time manner by the envisioned execution environment. Thus, a DEECo-based architecture can be understood as a distributed system of conditionally interacting feedback loops (the conditionality being given by the components' involvement in ensembles).

Comments on Authorship

My personal contribution to this paper lies in analyzing the advances in the area of SCEs and the related formal specification languages (i.e., SCEL [DNFLP13, DNLPT14]) and, under helpful guidance and supervision of the other authors, combining the promising ideas with traditional approaches to component-based architecture design. Specifically, I contributed with the idea of promoting the concept of component ensemble to a first-class architecture concept that represents a dynamic, attribute-based component binding and features stateless, data-flow-based, implicitly executed knowledge exchange. My contribution also includes the runtime-environment-driven computational model. Moreover, again under helpful guidance and supervision of the other authors, I authored a majority of the text.

Towards Dependable Emergent Ensembles of Components: The DEECo Component Model

Jaroslav Keznikl[1,2], Tomáš Bureš[1,2], František Plášil[1,2], Michal Kit[1]

[1] Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics, Charles University
Prague, Czech Republic
{keznikl, bures, plasil, kit}@d3s.mff.cuni.cz

[2] Institute of Computer Science
Academy of Sciences of the Czech Republic
Prague, Czech Republic
{keznikl, bures, plasil}@cs.cas.cz

***Abstract*—In the domain of dynamically evolving distributed systems composed of autonomous and (self-) adaptive components, the task of systematically managing the design complexity of their communication and composition is a pressing issue. This stems from the dynamic nature of such systems, where components and their bindings may appear and disappear without anticipation. To address this challenge, we propose employing separation of concerns via a mechanism of dynamic implicit bindings with implicit communication. This way, we strive for dynamically formed, implicitly interacting groups – ensembles – of autonomous components. In this context, we introduce the DEECo component model, where such bindings, as well as the associated communication, are managed in an automated way, enabling transparent handling of the dynamic changes in the system.**

Keywords—component; ensemble; adaptation; dynamic architecture; implicit communication; implicit bindings

I. Introduction

In component-based software architecture design, we still face many challenges, particularly in the case of large, distributed and dynamically changing applications, where both components and bindings may appear/disappear without anticipation. Therefore, components are often designed as autonomous [1] so that they stay operable regardless of the changes in their operating environment. This in turn implies the need for a (self-) adaptation mechanism [2].

In this context, a challenge is to find suitable paradigms for engineering such systems to overcome the design complexity of their communication and composition, specifically in terms of their autonomic and dynamic nature.

As for autonomy and (self-) adaptation, these have been partially addressed by agent-based approaches [3][4] where actors leveraging on messaging establish explicit bindings for data and code exchange. As for coping with dynamism, techniques utilizing implicit bindings while focusing on explicit communication have been proposed [5]. Furthermore, separation of concerns was to some extent achieved by introducing implicit communication (driven by a third-party entity) via explicit bindings [6]. Intuitively, it is desirable to combine all of these approaches in order to take advantage of the benefits they offer simultaneously.

Contributing to the ASCENS project [7], our goal is to respond to this challenge by elaborating on the idea of dynamic implicit bindings with implicit communication. To do so, we introduce the DEECo component model (Dependable Emergent Ensembles of Components).

The basic idea of DEECo is to facilitate separation of concerns by extracting component bindings and communication from the component implementation, expressing them implicitly, and managing them in an automated way via the DEECo runtime framework. Specifically, we consider bindings to be declaratively-expressed first-class entities, capturing component communication by implicit data exchange. This way, a component can be considered as an autonomous and self-aware entity, relying solely on its local data, which are modified in the background by the runtime framework according to the implicit component bindings. Similar to self-organizing architectures [8], such bindings facilitate dynamic forming of implicit groups – ensembles – of autonomous components.

Moreover, stemming from the need for autonomy while allowing for dependability, in DEECo we aim at supporting (self-) adaptation, code mobility, and verification of safety (reachability) properties.

The rest of this paper is structured as follows: Section II describes our motivating case study, in Section III the requirements for effectively addressing the outlined goals, demonstrated by the case study, are summarized, Section IV provides a brief description of the DEECo component model, while the concluding Section V outlines a long term DEECo vision and identifies the key challenges to be addressed.

II. Case Study

As our motivating case study we consider a robotic playground scenario, stemming from the e-mobility demonstrator [9] in ASCENS. Basically, it pertains to several autonomous robots moving on roads with crossings. When approaching a crossing, all the robots in the same situation, or already on the crossing, have to cooperate in order to avoid collision. An assumption is that the robots can communicate only with those within a short range, since they typically have limited communication signal coverage. Thus the architecture of the system of robots and crossings is dynamic, determined by their actual positions.

The work was partially supported by the EU project ASCENS 257414, the Grant Agency of the Czech Republic project P202/11/0312. The work was partially supported by Charles University institutional funding SVV-2012-265312.

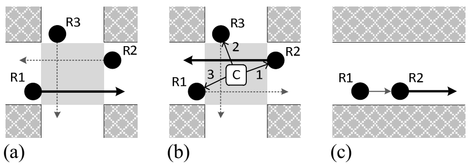


Figure 1. Robot Case Study: (a) autonomous robots, (b) robots advised by a crossing, (c) convoy

In the basic case (Fig. 1.a), we assume that the robots give priorities according to the "right-hand rule" (e.g., R1 has the highest priority). Furthermore, we consider also other (more elaborate) variants for the crossing strategy (Fig. 1.b), where the robots are advised by the crossing itself (similar to crossings with specific arrangements of traffic lights; e.g., the robot R2 is advised by C to continue as the first). These variants are handled by self-adaptation of the robots, including both short-term and permanent adaptation. As an example of the former case, the crossing provides the corresponding robots with a strategy for interacting with it only for the time the robots are at/in the crossing; in the latter case, robots exchange strategies for interacting with new variants of crossings, and these strategies are adopted permanently. Finally, we also expect the robots to form dynamic convoys (Fig. 1.c); i.e., if two robots drive in the same direction, one behind the other, the robot behind (e.g., R1) should adjust its speed to the one in front (e.g., R2). We will use this robot playground case study as the running example throughout this paper.

In addition to the robot scenario, we also seek inspiration from a more elaborate case study of a self-aware and self-adaptable cloud platform [9]. We consider several client applications running on a cloud platform, continuously storing their logging data via a logging service. An important part of the scenario is that the application processes, as well as the processes implementing the logging service, can migrate between the nodes of the cloud according to various optimization criteria. These processes should migrate autonomously and be able to adapt the migration strategy according to the impact of previous migrations. During migrations, client applications should not be affected by the changes in the cloud architecture.

III. Overview of Requirements

Based on the case studies, we have identified several general requirements to be met by the DEECo component model. These include the capability to:

- allow for convenient design with a suitable level of abstraction and proper concepts, coping efficiently with dynamic and parallel activities.
- provide appropriate means for continuous self-adaptation of the system. This implies the need for separation of concerns, so that adaptation is separated from business logic.
- achieve dynamic updates of behavior by means of both (self-) adaptation and code mobility.
- ensure a high level of dependability by supporting methods for formal verification of safety (reachability) properties.

As the requirements are also partially targeted by the SCEL [10] specification language proposed for ASCENS, we will reuse some of its related concepts. However, since SCEL is a low-level generic theoretical model, it does not provide any higher-level abstractions for system design. Supporting only low-level primitive operations for component communication without considering any programming environment, it is not, as such, suitable for direct development of non-trivial software systems.

IV. DEECo Component Model

In this section, we target the requirements identified in Section III by introducing the DEECo component model. Its basic idea is to manage the dynamism of a system by externalizing the (potentially distributed) communication among components. Specifically, a component accesses only its local data, which are communicated in the background to other components in an automated way by the DEECo runtime framework. Hence, a component is logically an autonomous unit, oblivious to the way data are exchanged, which makes it robust and adaptable with respect to dynamism. Moreover, the DEECo data exchange mechanism supports code mobility and adaptation.

A. Component Structure

A component is a unit of design and deployment, consisting of knowledge and processes.

Knowledge represents the internal state and functionality of the component. It is a hierarchical data structure, similar to a tuple space [10], mapping identifiers to (potentially structured) values. Values are either statically-typed data or functions; both being first-class entities. Only pure functions with no global variables are considered.

A *process*, being essentially a "thread", operates locally upon the knowledge by calling a specific function (being a part of the knowledge) to perform its task. Since global variables are disallowed, a process assigns a part of the knowledge to the actual parameters of the function (*input knowledge*), and on its completion updates a part of the knowledge (*output knowledge*) by the return value.

The example from Fig. 2 describes the component `Robot` (a singleton instance; multiple instances are expected to be created by cloning) in the DEECo DSL. It illustrates that a component is defined solely by its initial knowledge, which also syntactically contains the definition of the component's processes. Here, the `Robot` component's knowledge contains

```
component Robot = {
   id: RobotId = ”R1”;
   info: RobotInfo = {
      position: Position = { x = 1, y = 1};
      path: list Position = [];
   };
   otherRobots: map RobotId -> RobotInfo = {};
   stepf: fun(inout i: RobotInfo, in o: map RobotId->RobotInfo) = {
      ... };
   processes = {
      step: Process = {
         function = stepf;
         inputKnowledge=[info, otherRobots];
         outputKnowledge=[info];
         scheduling = PERIODIC(100ms);
}; }; };
```

Figure 2. Example of a DEECo component

```
interface IRobot = {
   id: RobotId;
   info: RobotInfo;
   otherRobots: map RobotId -> RobotInfo;
};
ensemble AutonomousRobotsEnsemble {
   member-interface: IRobot;
   coordinator-interface: IRobot;
   membership: fun(in r: IRobot, in c: IRobot, out ret: Boolean) = {
      ret = proximity(r.info.position, m.info.position) <= TRESHOLD;
   };
   coordinator-to-member: fun(inout m: IRobot, in c: IRobot) = {
      m.otherRobots=m.otherRobots.merge(c.otherRobots).except(m.id);
   };
   member-to-coordinator: fun(in m: IRobot, inout c: IRobot) = {
      c.otherRobots[m.id] = m.info;
}; };
```

Figure 3. Example of an ensemble prescription

the actual position of the robot and the list of remaining waypoints the robot has to drive through (`info`), and similar information about the robots in its close perimeter (`otherRobots`). The `Robot`'s only process `step` moves the robot (via the `stepf` function) by updating its `info.position` according to the `info.path` while considering `otherRobots` in order to avoid a crash (by applying the right-hand rule).

B. Component Composition

In DEECo, the composition of components is flat, in the form of component *ensembles* – groups of components, consisting of a single (unique) coordinator and multiple member components. At the same time, a component may play the role of a coordinator or member in several ensembles.

Supporting separation of concerns, an ensemble mediates communication between the coordinator and members. In consequence, two components can communicate only if they are involved in the same ensemble and one of them is the coordinator (direct communication among the members is not possible). Most importantly, such an involvement is expressed implicitly via a membership condition, evaluated in an automated way by the runtime framework.

Similarly, the inter-component communication is realized by implicit knowledge exchange (i.e., a part of the knowledge of one component is copied to the other component in an automated way). Such exchange may also include a knowledge transformation.

In compliance with the principle of knowledge exchange solely between the coordinator and a member, an ensemble is described pair-wise, defining the couples coordinator – member. Syntactically, an *ensemble prescription* consists of the desired knowledge *interface* of the coordinator (*coordinator interface*), the desired interface of a member (*member interface*), *membership function*, and *mapping function*.

Interface constitutes a structural prescription for a partial view on a component's knowledge. Specifically, it is associated with the knowledge by means of duck typing (structural subtyping); i.e., if a part of the component's knowledge matches the structure prescribed by the interface, then the component reifies the interface. For example, `Robot` from Fig. 2 reifies the `IRobot` interface from Fig. 3.

Membership function declaratively expresses the membership condition, under which two components form a pair

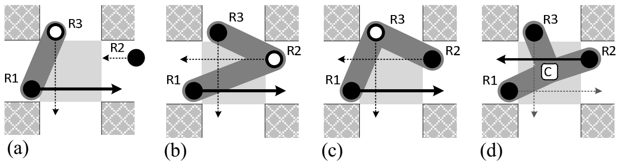


Figure 4. Ensemble Examples: (a) two-robot ensemble with coordinator R3, (b) autonomous robots ensemble with coordinator R2, (c) autonomous robots ensemble with coordinator R3, (d) crossing ensemble

coordinator – member of the ensemble. This condition is defined upon the knowledges of the components and is to be evaluated by the runtime framework (potentially in a distributed fashion). For example, in Fig. 3 the components `r` and `c`, reifying the `IRobot` interface, have to be in the proximity lower than `THRESHOLD` in order to form a coordinator-member pair.

Mapping function determines the knowledge exchange between the coordinator and a member. Specifically, it describes which part of the knowledge of one component is to be transferred to the other and how it is potentially transformed. We assume a separate mapping for each of the directions, coordinator-to-member and member-to-coordinator. Also, the mapping function is to be executed by the runtime framework. This basically ensures that relevant knowledge changes in one component are propagated to the other in the background. As an example, consider the `coordinator-to-member` and `member-to-coordinator` mapping functions from Fig. 3 which ensure an exchange of knowledge necessary to avoid robot collisions (i.e., the positions and remaining paths of the robots in a close perimeter).

In general, components form an ensemble whenever they satisfy the *ensemble condition* of an ensemble prescription, i.e., one of them reifies the coordinator interface, the other components reify the member interface, and the membership condition holds for each coordinator – member pair. Therefore, multiple ensembles based on the same prescription can be formed simultaneously.

As an example, consider an ensemble prescription of autonomous robots where the membership condition requires the member robots to be in close proximity to the coordinator robot. In Fig. 4.a, R2 is too far from the coordinator R3 so it is not (yet) included in the ensemble [R1, R3]. After R2 reaches the required proximity, all three robots will form a single ensemble as shown in Fig. 4.b and Fig. 4.c (bigger ensembles are preferred to smaller ones and the coordinator is selected randomly if multiple candidates are eligible). Assuming the crossing strategy, where components are advised by the crossing, the ensemble will potentially look like the one in Fig. 4.d, where the crossing component is the coordinator.

In the situation where a component satisfies the ensemble condition for multiple ensembles (Fig. 4.b and Fig. 4.c), we envision a mechanism for deciding whether all or only a subset of the candidate ensembles should be formed. Currently, we employ a mechanism based on a partial order over ensembles (the ensemble with higher order is preferred; incomparable ensembles are formed simultaneously).

C. Computational Model

The computational model of DEECo is based on asynchronous knowledge exchange and process execution, stemming from the asynchronous nature of dynamic distributed systems. Specifically, the processes of all components execute in parallel as independent threads either periodically or when triggered by modification of (a part of) their input knowledge. In a similar vein, a binding in an ensemble is accomplished by a separate activity, running the mapping function again either periodically or when triggered by a change in the knowledge of the coordinator/member.

Due to the asynchrony, it is necessary to ensure that knowledge is accessed consistently. Thus, at its start, a process is atomically provided with a copy of its input knowledge so that its computation is not affected by later-occurring knowledge modifications. When finishing, the process atomically updates its output knowledge. The same atomic copy-on-start and update-on-return semantics also applies to the membership and mapping functions of ensembles. Technically, this semantics can be implemented for instance via messaging.

For the time being, we envision employing the "single writer, multiple readers" rule for knowledge access, meaning that at any time each value in the knowledge of a component has at most one writer while being accessed by potentially multiple readers. Note that this rule applies to obtaining input and writing output knowledge of component processes, as well as to knowledge exchange via mapping functions. Since all the readers and writers are well defined, we envision that compliance with this rule will be verified.

Consequently, based on the computational model, an ensemble is created when the ensemble condition starts to hold, and is discarded when the condition gets violated. Technically, as the whole system is asynchronous and potentially distributed, techniques for handling inherent delays, while creating/discarding ensembles, have to be carefully chosen.

V. Discussion: Vision and Challenges

We assume DEECo will be employed in the design of systems of autonomous self-adaptive components, such as a self-managing cloud platform and self-organizing car sharing [9], where it aims at simplifying the design process.

Specifically, we expect DEECo to effectively handle knowledge exchange among distributed components, including code mobility in support of adaptation, while putting a strong emphasis on separation of concerns. Although similar to software connectors [11], DEECo ensembles capture component composition implicitly and thus allow for handling of dynamic changes in an automated way. Similar benefits result from the implicit knowledge exchange.

Currently, we foresee two possible methods for handling distributed knowledge exchange: message passing and distributed tuple spaces, both already adopted by the state-of-the-art agent-oriented frameworks such as [3] and [12], respectively. Although supporting dynamic features such as code mobility, these frameworks lack high-level abstractions allowing for implicit dynamic composition and communication. Nevertheless, since DEECo components resemble agents with respect to autonomy, we consider partially employing these frameworks in the DEECo runtime framework. Currently, we already have prototypes for both types of these methods for handling knowledge exchange[1].

In order to support controlled architecture evolution, we aim to incorporate mechanisms for dynamic addition, modification, and removal of ensemble prescriptions.

In addition, we envision supporting formal verification of DEECo applications. As for model checking of temporal properties, we assume a mapping of applications to SCEL and intend to exploit its means [12] for this purpose. Moreover, we anticipate also employing stochastic model checking [13][14] for quantitative verification.

Finally, inspired by the cloud and e-mobility case studies, we intend to introduce, in addition to abstractions for performance awareness, other forms of implicit knowledge-based communication such as distributed consensus.

References

[1] J. O. Kephart, and D. M. Chess, "The vision of autonomic computing," Computer, vol. 36, IEEE CS, 2003, pp. 41-50.

[2] R. N. Taylor, N. Medvidovic, and P. Oreizy, "Architectural styles for runtime software adaptation," Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA 2009), 2009, pp. 171–180.

[3] F. Bellifemine, G. Caire, and D. Greenwood, "Developing multi-agent systems with Jade," John Wiley & Sons, 2007.

[4] E. Gjondrekaj, M. Loreti, R. Pugliese, and F. Tiezzi, "Modeling adaptation with a tuple-based coordination language," Proc. of 27th Symposium on Applied Computing (SAC 2012), 2012, in press.

[5] C. Escoffier and R. S. Hall, "Dynamically adaptable applications with iPOJO service, " Software Composition, 2007.

[6] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in BIP," Proc. of Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06), 2006, pp. 3-12.

[7] ASCENS [Online], http://www.ascens-ist.eu.

[8] C. Villalba, M. Mamei, and F. Zambonelli, "A self-organizing architecture for pervasive ecosystems," Self-Organizing Architectures, volume 6090 of LNCS, pp. 275–300, 2010.

[9] N Serbedzija, S. Reiter, M. Ahrens, J. Velasco, C. Pinciroli, N. Hoch, and B. Werther, "Requirement specification and scenario description," ASCENS Deliv. D7.1, November 2011.

[10] R. De Nicola, G. Ferrari, M. Loreti , and R.Pugliese, "Languages primitives for coordination, resource negotiation, and task description," ASCENS Deliv. D1.1, 2011, http://rap.dsi.unifi.it/scel/.

[11] R.N. Taylor, N. Medvidovic, and E.M. Dashofy: "Software architecture: foundations, theory, and practice," Wiley, 2010.

[12] L. Bettini et al. "The Klaim project: theory and practice," In global computing. Programming Environments, Languages, Security, and Analysis of Systems, volume 2874 of LNCS, 2003, pp. 88-150.

[13] M. Z. Kwiatkowska, G. Norman, D. Parker, and H. Qu, "Assume-guarantee verification for probabilistic systems," Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2010), Springer, 2010, pp. 23-37.

[14] J. Barnat, L. Brim, I. Cerna, M. Ceska, and J. Tumova: "ProbDiVinE, a parallel qualitative LTL model checker," Quantitative Evaluation of Systems (QEST 07), IEEE, 2007.

[1] http://d3s.mff.cuni.cz/projects/components_and_services/deeco/

3.2 DEECo: an Ensemble-Based Component System

Tomáš Bureš,
Ilias Gerostathopoulos,
Petr Hnětynka,
Jaroslav Keznikl,
Michal Kit,
František Plášil

In proceedings of the **16th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE '13)**.

Published by ACM,
pages 81-90,
ISBN 978-1-4503-2122-8,
June 2013.

The original version is available electronically from the publisher's site at http://dx.doi.org/10.1145/2465449.2465462.

Summary of the Paper

This paper, published as [BGH+13], forms the central part of this thesis by focusing on all the three research goals **G1-G3** presented in Section 1.4 (considering to an extent all the challenges outlined in Section 1.3). It builds on the vision described in Section 3.1 and presents a fully elaborated version of the DEECo component model [8], including the corresponding execution environment and development process.

In particular, the paper advocates employing the component-based software engineering perspective, which brings design-time scalability via well-defined architecture design, while, at the same time, exploiting the specifics of DEECo to deal with the challenges of dynamicity, open-endedness, robustness, and autonomicity of RDS. To this end, the paper presents DEECo as an instantiation of a thereby introduced class of component-based systems – *Ensemble-Based Component Systems* (EBCS), defined as "*Distributed systems composed of components that feature autonomic and (self-) adaptive behaviors and are organized into emergent ensembles to achieve cooperation*." EBCS (including DEECo) are viewed as a synergy of component-based software engineering, agent-oriented computing, ensemble-oriented systems, and control system engineering.

Contributing to the research goal **G1,** the paper presents an elaborate description of the DEECo architecture abstractions and shows that the DEECo component model well combines the encapsulation, composability, and reusability, brought by component-based software architectures, with the needs of autonomic behavior and extreme architecture dynamism in RDS. The paper also promotes the explicit notion of *belief* as the part of a component's knowledge that represents the component's view of the environment and the other components. Belief is updated in an asynchronous, best-effort manner via ensemble knowledge exchange, and is thus treated with a certain level of uncertainty as it might become obsolete or invalid.

A major part of the paper focuses on attacking the research goal **G3**. To this end, the paper outlines a formalized computational model of DEECo emphasizing distributed execution. While exploiting the stateless, best-effort nature of ensemble membership and knowledge exchange, the computational model allows for various specializations, offering different tradeoffs between decentralization and performance. Due to space constraints, the formalization of the computational model is referred via a technical report [AABG+13]. Further, the paper presents a realization of this computational model in Java – jDEECo [13]. jDEECo includes a mapping of the DEECo abstractions (i.e., components and ensembles) to Java language primitives in terms of an internal DSL based on Java annotations. This way, the mapping does not require any language extensions or external tools. The core of jDEECo is the Java implementation of the execution environment and the corresponding tool support. The jDEECo execution environment is primarily responsible for scheduling component processes, forming ensembles, and performing knowledge exchange. It also allows for distribution of components. The execution environment is internally divided into (i) a management layer, responsible for scheduling and execution of component processes and ensemble knowledge exchange within regular Java threads, and (ii) knowledge repository, storing and handling access to component knowledge. The implementation described in the paper offers two variants of knowledge repository: a local one, employed for efficient simulation and verification of jDEECo applications, and a distributed one, which is based on the JavaSpaces [2] middleware and is used when the execution environment needs to run in a distributed setting. Hence, the distribution is achieved at the level of knowledge repository. (A fully decentralized implementation is subject to ongoing research [BGH+14a].)

Although primarily focusing on distributed execution, autonomy, and dynamism, jDEECo also facilitates dependability by supporting formal analysis of DEECo-based applications at the level of implementation (i.e., challenge C4 in the context of the research goal **G3**). Specifically, it is integrated with Java PathFinder [VHB+03][16] so as to enable verification of knowledge-related LTL properties. In this context, the relatively restrictive computational model of DEECo and a PathFinder-optimized implementation of the execution environment allowed making the verification efficient.

Finally, tackling the research goal **G2**, the paper elaborates on the component-based development (CBD) process, while focusing on traceability between system requirements and architecture. Specifically, the paper advocates integration of the traditional CBD process with the *Invariant Refinement Method* (IRM) – a requirements-driven, formally-grounded method for designing DEECo-based architectures, presented separately in Section 3.3. The main goal of IRM is identification of DEECo components and ensembles by systematic decomposition and refinement of system requirements. This subsequently brings correct-by-construction guarantees of compliance with system requirements and the possibility of automated preparation of DEECo artifacts (e.g., component skeletons, ensemble definitions). The CBD process, on the other hand, builds on separation of system development process from component development process. The integrated development process proposed in the paper features requirements analysis and high-level architecture design phases that are based on IRM, followed by separate phases of component and ensemble development, concluded by traditional integration, testing, and maintenance phases of the CBD process. The synergy of IRM with the CBD process complements DEECo at design time by enabling the system-level design view of IRM to be readily translated to a DEECo-based implementation. Moreover, the traceability between system requirements and architecture provided by IRM is advantageously exploited during component and system development as a baseline for unit and integration testing, respectively.

The contributions of the paper are illustrated and evaluated on the electrical vehicle navigation case study featured by the ASCENS project [SHP+13, SRA+11].

Comments on Authorship

Overall, the paper is of equal authorship. I specifically participated on the elaboration of the DEECo abstractions and their realization in terms of jDEECo, as well as on the corresponding evaluation in terms of the electrical vehicle navigation case study. I also significantly participated on the elaboration and formalization of the DEECo computational model. Finally, I was responsible for enabling formal analysis of DEECo-based applications via the integration with Java PathFinder.

DEECo – an Ensemble-Based Component System

Tomas Bures[1,2]
bures@d3s.mff.cuni.cz

Ilias Gerostathopoulos[1]
iliasg@d3s.mff.cuni.cz

Petr Hnetynka[1]
hnetynka@d3s.mff.cuni.cz

Jaroslav Keznikl[1,2]
keznikl@d3s.mff.cuni.cz

Michal Kit[1]
kit@d3s.mff.cuni.cz

Frantisek Plasil[1]
plasil@d3s.mff.cuni.cz

[1]Charles University in Prague
Faculty of Mathematics and Physics
Prague, Czech Republic

[2]Institute of Computer Science
Academy of Sciences of the Czech Republic
Prague, Czech Republic

ABSTRACT

The recent increase in the ubiquity and connectivity of computing devices allows forming large-scale distributed systems that respond to and influence activities in their environment. Engineering of such systems is very complex because of their inherent dynamicity, open-endedness, and autonomicity. In this paper we propose a new class of component systems (*Ensemble-Based Component Systems* – EBCS) which bind autonomic components with cyclic execution via dynamic component ensembles controlling data exchange. EBCS combine the key ideas of agents, ensemble-oriented systems, and control systems into software engineering concepts based on autonomic components. In particular, we present an instantiation of EBCS – the DEECo component model. In addition to DEECo main concepts, we also describe its computation model and mapping to Java. Lastly, we outline the basic principles of the EBCS/DEECo development process.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems – d*istributed applications*; D.2.6 [**Software Engineering**]: Programming Environments – i*ntegrated environments*; D.2.9 [**Software Engineering**]: Management – l*ife cycle*; D.2.11 [**Software Engineering**]: Software Architectures.

Keywords

Component model; emergent architecture; component ensembles; autonomic systems; development process; runtime framework

1. INTRODUCTION

The significant increase in the ubiquity and connectivity of computing devices has opened new possibilities for addressing social and environmental challenges (e.g., ambient assisted living, smart city infrastructures, emergency coordination, environmental monitoring) by providing hardware and infrastructures necessary for building large-scale Resilient Distributed Systems (RDS) that respond to and influence activities in the real world. As RDS have to cope with very dynamic and open-ended environments, they exhibit a high degree of adaptivity and autonomicity.

Although developing RDS has become relatively feasible from the perspective of hardware and network infrastructures, there still remain significant challenges in developing software for RDS. In particular, the problem is to feature the appropriate computation models and development processes which would address the requirements of scalability, distribution, and well-defined architecture, while, at the same time, would deal with the requirements of dynamicity, open-endedness, robustness, and autonomicity.

1.1 Towards EBCS

In this paper, we advocate using components for engineering RDS. The use of components has been proven efficient for the design and development of large-scale systems with well-defined architectures. However, due to the dynamic and autonomic nature of RDS, traditional approaches to component architectures [38] as well as existing component models [6][7][30][31][32] do not scale. Therefore, inspired by the work in the field of formal coordination languages [14], in this paper we address this issue by identifying a new class of component-based systems – *Ensemble-Based Component Systems* (EBCS) – specifically tailored for designing RDS. Moreover, we present the DEECo (Distributed Emergent Ensembles of Components) component model [8][25] as our instantiation of EBCS.

The characteristic of EBCS is that the "traditional" explicit component architecture is replaced by the composition of components into so-called *ensembles* [14][20], each of which is an implicit, inherently dynamic group of components mutually cooperating to achieve a particular goal. To cope with the dynamism, the components in EBCS become autonomic entities, building on agent-oriented concepts [39], while featuring execution model based on feedback loops (e.g., MAPE-K [23], soft real-time control systems [33]) in order to achieve (self-) adaptive and resilient operation.

In this view, the EBCS can be defined as "*Distributed systems composed of components that feature autonomic and (self-) adaptive behaviors and are organized into emergent ensembles to achieve cooperation.*"

EBCS thus naturally combine relevant concepts from a number of research areas (Figure 1). Namely:

From *component-based software engineering* [11] EBCS adopt the software engineering concepts of the system architecture based on components (which themselves are seen as well-encapsulated, reusable, and substitutable entities) and the component-based development process.

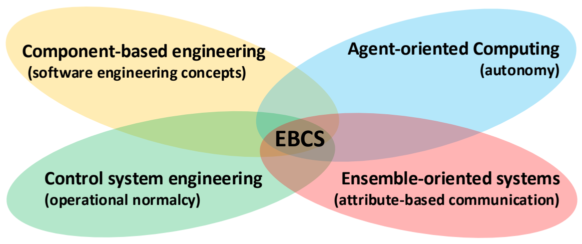

Component-based engineering
(software engineering concepts)
Agent-oriented Computing
(autonomy)
EBCS
Control system engineering
(operational normalcy)
Ensemble-oriented systems
(attribute-based communication)

Figure 1: Areas combined into Ensemble-Based Component Systems and their strong points.

From *agent-oriented computing* [39] EBCS derive the autonomous aspects, where the individuals maintain only a partial view on the whole system in order to guide their decisions – the belief, and self-* behavior [10]. This way, the overall behavior of EBCS is an emergent result of the behaviors of the individual components, enabling thus for efficient decentralized execution.

Building on the *ensemble-oriented systems* [14][20] EBCS rely on the attribute-based communication, which models the communication as best-effort and localized to dynamically changing ensembles of components; as opposed to existing agent-based systems [4] which at the deployment level resemble service-oriented architectures employing explicit communication channels. This helps to effectively cope with the assumption that the deployment (and thus also architecture) of RDS changes very dynamically.

From *control system engineering* [33] EBCS adopt the idea of achieving robustness by employing (soft real-time) control feedback loops [23] that maintain the *operational normalcy* of a component. Here, operational normalcy refers to the property of being within certain limits that define the range of normal functioning of the component. The required level of robustness is achieved by adjusting the periods of the loops. As extreme dynamism is assumed, the core attribute of EBCS is employing the concept of feedback loops both at the level of individual components and ensembles. Thus, an EBCS-based system can be understood as a distributed system of conditionally interacting feedback loops.

As a result, EBCS provide the following key features important for development of RDS:

- System architecture (represented by components and their bindings) *emerges* at runtime. The system architecture is however not arbitrary – it complies with explicit interaction patterns of ensembles specified at design time.
- Components maintain a *belief* about the rest of the system and the environment. The belief is managed outside the component behavior by the underlying runtime framework.
- Component execution is performed *in isolation* based solely on the component's belief. This strengthens the autonomicity of components (e.g., in the context of unreliable communication and/or rapid architecture changes).

1.2 Goals and Structure of the Text

The goal of the paper is to describe our instance of EBCS – the DEECo (Distributed Emergent Ensembles of Components) component model [8][25] and its framework – and to share with the reader our experience with its application.

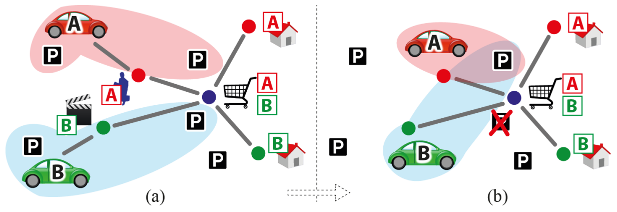


Figure 2: E-mobility: Potential ensembles and their dynamic changes (available parking stations close to respective POIs).

In particular, after describing a running example (Section 2), we present: (i) the core DEECo concepts along with its abstract execution model (Section 3), (ii) a Java-based DEECo framework, which allows engineering DEECo components and ensembles in a Java environment (Section 4), and (iii) an outline of a design process, which drives the architectural design of EBCS (DEECo-based systems in particular) from high-level requirements (Section 5). Finally, we share with the reader our experience with an industrial case study (Section 6). After presenting a survey of related work (Section 7), the paper concludes with a summary and a brief overview of our intentions in future work (Section 8).

2. RUNNING EXAMPLE

We illustrate the main concepts of EBCS/DEECo with the help of the electrical vehicle navigation case study featured by the ASCENS project [37]. We describe the fundamentals of the case study in this section and articulate the running example that we use in the rest of the paper.

The objective of the e-mobility case study is to coordinate the planning of journeys in compliance with parking and charging strategies in a highly dynamic and heterogeneous traffic environment, where information is distributed. The case study consists of drivers, navigating around a city in their electric vehicles (*e-vehicles*). Drivers have to reach particular *Points Of Interest* (POIs) within time constraints, specified as the expected POI arrival and departure times. Every driver possesses his/her daily meetings schedule (*calendar*), where POIs and their respective constraints are listed. Vehicles are equipped with sensors of basic capabilities, e.g., monitoring the battery level and energy consumption of the car, but also more sophisticated ones, e.g., monitoring the traffic level along the route. Vehicles can only park and recharge in designated parking spaces and charging lots, organized into parking/charging stations. They also communicate with each other and with relevant parking/charging stations, e.g. those that are close to their respective POIs. Such communication is necessary, e.g., in order for a vehicle to obtain the availability of the parking station and potentially reserve a place there. It is important that in this setting no central coordination point is assumed; there is no global control or global planning. Instead, every e-vehicle plans and executes its route individually, based on the data available.

The whole system can be seen as a set of (distributed) nodes, which form ensembles (dynamic communication groups) in order to allow drivers to arrive at their POIs in time while leveraging the available resources in a close-to-optimal way. This is illustrated in Figure 2 – each vehicle forms an ensemble with available parking stations close to their respective POIs. Figure 2.b further shows an evolution of the scenario, where vehicles have moved along the route and a parking station has become unavailable leading to dynamic changes of the ensembles.

As our running example, we consider a simplified version of the case study by making the following assumptions: i) car sharing is not allowed, so drivers are bound to the vehicles they drive, ii) parking and charging stations are modeled together as Parking Lot/Charging Station (PLCS) elements, iii) drivers do not reserve a place in the PLCSs, but only obtain their availability information in order to plan accordingly, and iv) PLCSs are relevant w.r.t. a vehicle if they are within a fixed distance to one of the vehicle's POIs.

Although simplified, the running example features a number of important challenges targeted by EBCS. In particular, the physical architecture of the system constantly changes as the cars move around the city; cars and PLCSs maintain a partial view over the whole system, according to the information they obtain from components they interact with; trip planning and decision making in general is localized to the drivers (cars), as no central coordination is assumed.

3. DEECo COMPONENT MODEL

To refine the principles of EBCS into a systematic approach for building software for RDS, we have proposed a new component model called DEECo [25]. DEECo embodies the main concepts of EBCS, while giving them a suitable semantics in order to turn them into proper software engineering constructs that can be employed in the real-life development of RDS.

3.1 General Concepts

DEECo is built on top of two first-class concepts: *component* and *ensemble.* A component is an independent and self-sustained unit of development, deployment and computation. An ensemble acts as a dynamic binding mechanism, which links a set of components together and manages their interaction. A grounding idea in DEECo is that the only way components bind and communicate with one another is through ensembles. The two first-class DEECo concepts are in detail elaborated below. An integral part of the component model is also the runtime framework providing the necessary management services for both components and ensembles.

3.1.1 Component

A component in DEECo comprises *knowledge*, exposed via a set of *interfaces*, and *processes*, as illustrated in Figure 3.

Knowledge reflects the state and available functionality of the component (lines 8-16). It is organized as a hierarchical data structure (resembling a tuple space [15]), which maps knowledge identifiers to values. Specifically, values may be either potentially structured data or executable functions. Technically, we use structured identifiers to refer to internal parts of the structured values (e.g., plan.isFeasible in line 18). In this context, the term belief refers to the part of a component's knowledge that represents a copy of knowledge of another component, and is thus treated with a certain level of uncertainty as it might become obsolete or invalid.

A component's knowledge is exposed to the other components and environment via a set of interfaces (lines 7, 29). An interface (e.g., lines 1-2) thus represents a partial view on the component's knowledge. Specifically, interfaces of a single component can overlap and multiple components can provide the same interface, thus allowing for polymorphism of components.

Component processes are essentially soft real-time tasks that manipulate the knowledge of the component. A process is characterized as a function (lines 19-21) associated with a list of input and output knowledge fields (line 18). Operation of the process is managed by the runtime framework and consists of atomically retrieving all input knowledge fields, computing the process function, and atomically writing all output knowledge fields. A process may have side effects in terms of sensing and actuating, however, it is not supposed to explicitly communicate with other components or other processes of the same component in any other way than via knowledge.

```
1.  interface AvailabilityAggregator:
2.     calendar, availabilities
3.
4.  interface AvailabilityAwareParkingLot:
5.     position, availability
6.
7.  component Vehicle features AvailabilityAggregator:
8.     knowledge:
9.        batteryLevel = 90%,
10.       position = GPS(…),
11.       calendar = [ POI(WORKPLACE, 9AM-1PM), POI(MALL, 2PM-3PM), … ],
12.       availabilities = [ ],
13.       plan = {
14.             route = ROUTE(…),
15.             isFeasible = TRUE
16.       }
17.    process computePlan:
18.          in plan.isFeasible, in availabilities, in calendar, inout plan.route
19.       function:
20.          if (!plan.isFeasible)
21.             plan.route ← Planner.computePlan(calendar, availabilities)
22.       scheduling: triggered( changed(plan.isFeasible) ∨ changed(availabilities) )
23.    process checkPlanFeasibility:
24.          in plan.route, in batteryLevel, in position, out plan.isFeasible
25.       function:
26.          plan.isFeasible ← Planner.isFeasible(plan.route, batteryLevel, position)
27.       scheduling: periodic( 5000ms )
28.
29. component PLCS features AvailabilityAwareParkingLot:
30.    knowledge:
31.       position = GPS(…) ,
32.       availability = …
33.    process observeAvailability:
34.          out availability
35.       function:
36.          availability← Sensors.getCurrentAvailability()
37.       scheduling: periodic( 2000ms  )
```

Figure 3: Examples of DEECo component definitions in a DSL

Being active entities of computation implementing feedback loops, component processes are subject to cyclic scheduling, which is again managed by the runtime framework. A process can be scheduled either periodically (line 27), i.e., repeatedly executed once within a given period, or as triggered (line 22), i.e., executed when a trigger condition is met. For brevity, we assume the change of input knowledge value as the only trigger condition.

Referring to the e-mobility running example, the components (each occurring in multiple instances) are the Vehicle and the PLCS (Figure 3). A Vehicle maintains a belief over the availability of the relevant PLCSs (availabilities, line 12). It uses a Planner library to (re-) compute its journey plan according to the availability belief and its calendar (line 17) every time the availability belief or plan feasibility changes (line 22). The Vehicle also periodically checks if its plan remains feasible, with respect to its battery level and its current position (line 23). A PLCS just keeps track of its available timeslots for vehicle parking and charging (lines 33-37).

3.1.2 Ensemble

An ensemble embodies a dynamic binding among a set of components and thus determines their composition and interaction. In DEECo, composition is flat, expressed implicitly

```
1.   ensemble UpdateAvailabilityInformation:
2.     coordinator: AvailabilityAggregator
3.     member: AvailabilityAwareParkingLot
4.     membership:
5.         ∃ poi ∈ coordinator.calendar:
6.             distance(member.position, poi.position) ≤ TRESHOLD &&
7.             isAvailable(poi, member.availability)
8.     knowledge exchange:
9.         coordinator.availabilities ← { (m.id, m.availability) | m ∈ members }
10.    scheduling: periodic( 5000ms )
```

Figure 4: An example of an ensemble definition in a DSL.

via a dynamic involvement in an ensemble. Among the components involved in an ensemble, one always plays the role of the ensemble's *coordinator* while others play the role of the *members*. This is determined dynamically (the task of the runtime framework) according to the *membership* condition of the ensemble. As to interaction, the individual components in an ensemble are not capable of explicit communication with the others. Instead, the interaction among the components forming the ensemble takes the form of *knowledge exchange*, carried out implicitly (by the runtime framework, Section 4.2).

Specifically, definition of an ensemble (Figure 4) consists of:

- *Membership condition.* Definition of a membership condition includes the definition of the interface specific for the coordinator role – *coordinator interface* (line 2), as well as the interface specific for the member role (and thus featured by each member component) – *member interface* (line 3), and the definition of a *membership predicate* (lines 4-7). A membership predicate declaratively expresses the condition under which two components represent a coordinator-member pair of the associated ensemble. The predicate is defined upon the knowledge exposed via the coordinator/member interfaces and is evaluated by the runtime framework when necessary. In general, as illustrated in Figure 5, a single component can be member/coordinator of multiple ensembles, so that ensembles form overlapping composition layers upon the components.
- *Knowledge exchange.* Knowledge exchange embodies the interaction between the coordinator and all the members of the ensemble (lines 8-9); i.e., it is a one-to-many interaction (in contrast to the one-to-one form of the membership predicate). Being limited to coordinator-member interaction, knowledge exchange allows the coordinator to apply various interaction policies. In principle, knowledge exchange is carried out by the runtime framework; thus, it is up to the runtime framework when/how often it is performed. Similarly to component processes, knowledge exchange can be carried out either periodically or when triggered (line 10).

Based on the ensemble definition, a new ensemble is dynamically formed for each group of components that together satisfy the membership condition.

In summary, each component operates only upon its own local knowledge, which gets implicitly updated by the runtime framework (via knowledge exchange) whenever the component is part of an ensemble. This supports component encapsulation and independence. Further details are elaborated in [2].

The sole ensemble of the running example is the UpdateAvailabilityInformation ensemble listed in Figure 4. Its purpose is to aggregate the availability information of the members, i.e. PLCSs, on the side of the coordinator, i.e., Vehicle (line 9). The ensemble is formed only when a PLCS is close enough to at least one of the POIs of the Vehicle (line 6) and there

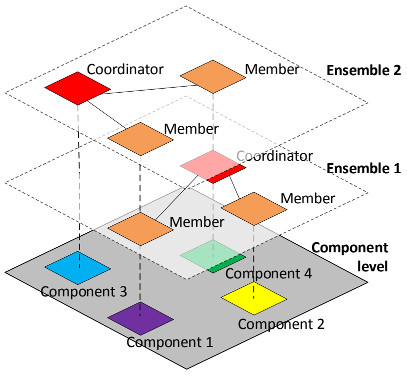


Figure 5: Composition of components into multiple overlapping ensembles in DEECo.

is an available slot in the PLCS, which can accommodate the respective POI arrival and departure time (line 7).

3.2 Computational Model

To allow for formal reasoning about DEECo applications, we have defined the operational semantics of DEECo, which models a DEECo application as a label transition system (LTS) with knowledge manipulation actions on transitions. The semantics further associates time with the LTS run and defines periodic and triggered processes and ensembles in terms of time constraints over traces generated by the LTS.

We also define a subset relation over a set of traces of observable changes in the components' knowledge. This allows us to build different implementations of DEECo (such as the tuple-space based implementation described in Section 4 and a messaging-based implementation following the protocol outlined in [2]) while accommodating for and benefiting from the specifics of the communication middleware used.

Due to space constraints we do not include the definition of the semantics in this paper, rather we refer the reader to the technical report [2], which describes it in full extent.

4. DEECo REALIZATION IN JAVA

In order to bring DEECo abstractions to the practical use during the development of real-life RDS we provide a framework called jDEECo [13], which is a Java-based realization of DEECo component model. jDEECo delivers the necessary programming abstractions and the runtime environment to deploy and run DEECo-based applications.

In this section, we describe how jDEECo maps definitions of DEECo components and ensembles to Java language primitives. In particular, we follow the developer's perspective and show how the running example gets implemented using the jDEECo constructs. Further, we briefly discuss interesting aspects of the jDEECo runtime framework and supporting tools and the in-memory representation of the DEECo concepts.

4.1 Mapping of DEECo Concepts to Java

By building on Java annotations, the mapping of DEECo concepts relies on standard Java language primitives and does not require any language extensions or external tools.

4.1.1 Component

A component definition has the form of a Java class (Figure 6). Such a class is marked by the @DEECoComponent annotation and extends the ComponentKnowledge class. The initial knowledge

```
1.  @DEECoComponent
2.  public class Vehicle extends ComponentKnowledge {
3.
4.      public List<CalendarEvent> calendar;
5.      public Plan plan;
6.      public EnergyLevel batteryLevel;
7.      public Map<ID, Availability> availabilities;
8.      public Position position;
9.
10.     public Vehicle() {
11.         // initialize the initial knowledge structure reflected by the class fields
12.     }
13.
14.     @DEECoProcess
15.     public static void computePlan(
16.         @DEECoIn("plan.isFeasible") @DEECoTriggered Boolean isPlanFeasible,
17.         @DEECoIn("availabilities ") @DEECoTriggered Map<…> availabilities,
18.         @DEECoIn("calendar") List<CalendarEvent> calendar,
19.         @DEECoInOut("plan.route") Route plannedRoute
20.     ) {
21.         // re-compute the vehicle's plan if it's infeasible
22.     }
23.
24.     @DEECoProcess
25.     @DEECoPeriodicScheduling(5000)
26.     public static void checkPlanFeasibility(
27.         @DEECoIn("plan.route") Route plannedRoute,
28.         @DEECoIn("batteryLevel") EnergyLevel batteryLevel,
29.         @DEECoIn("position") Position position,
30.         @DEECoOut("plan.isFeasible") OutWrapper<Boolean> isPlanFeasible
31.     ) {
32.         // determine feasibility of the plan
33.     }
34.     ...
35. }
36. public class Plan extends Knowledge {
37.     public Route route;
38.     public Boolean isFeasible;
39. }
```

Figure 6: Example of a component definition in Java.

structure of the component is captured by means of the public, non-static fields of the class (lines 4-8). The id knowledge field, which is used for unique identification of a component, is inherited from the ComponentKnowledge class. As knowledge can be hierarchically structured, these fields represent the first level of this hierarchy, where each can take the form of a knowledge tree (recursively), map, or list. As for the knowledge tree form, the non-leaf nodes of this tree need to be instances of a class inheriting from Knowledge (lines 36-39). The non-structured knowledge values are represented as serializeable Java objects. At runtime, this initial knowledge structure is initialized either via static initializers or via the constructor of the class (lines 10-12).

For convenience, the set of supported interfaces is implicit; i.e., all interfaces that structurally match the component's knowledge are assumed to be featured by the component (similar to duck typing in dynamic languages).

The component processes are defined as public static methods of the class, annotated with @DEECoProcess (e.g., lines 14-22). The requirement of the static modifier stems from the semantics of component process execution (Section 3.1.1). In particular, except for reading the input knowledge and writing the output knowledge (which is anyway managed by the runtime framework), a component process executes in isolation, without access to the knowledge. Thus, declaring the method as static prevents it from directly accessing the initial knowledge represented by the class fields (which are non-static).

The input and output knowledge of the process is represented by the methods' parameters. The parameters are marked with one of

```
1.  @DEECoEnsemble
2.  @DEECoPeriodicScheduling(4000)
3.  public class UpdateAvailabilityInformation extends Ensemble {
4.
5.      @DEECoEnsembleMembership
6.      public static boolean membership (
7.          @DEECoIn("coord.calendar ") List<CalendarEvent> calendar,
8.          @DEECoIn("member.position ") Position plcsPosition,
9.          @DEECoIn("member.availability ") Availability availability
10.     ) {
11.         for (CalendarEvent ce : eventsCalendar) {
12.             if (isClose(ce.poi.position, plcsPosition, DISTANCE_THRESHOLD)
13.                 && isAvailable(ce.poi, availability))
14.                 return true;
15.         }
16.         return false;
17.     }
18.
19.     @DEECoEnsembleKnowledgeExchange
20.     public static void knowledgeExchange (
21.         @DEECoIn("coord.calendar") List<CalendarEvent> calendar,
22.         @DEECoInOut("coord. availabilities") Map<…> availabilities,
23.         @DEECoIn("member.id]") ID memberID,
24.         @DEECoIn("member.position") Position plcsPosition,
25.         @DEECoIn("member.availability") Availability availability
26.     ) {
27.         availabilities.put (memberID, availability.clone(currentTimestamp()));
28.     }
29. }
```

Figure 7: Example of an ensemble definition in Java.

the annotations @DEECoIn, @DEECoOut or @DEECoInOut, in order to distinguish between input and output knowledge fields of the process (e.g., lines 16-19). Each annotation also includes an identifier of the knowledge field that the associated method parameter represents. As the input/output knowledge can consist of a knowledge field that is an internal node of a knowledge tree, the identifier of such a knowledge field is a dot-separated representation of the path to the node in the tree (e.g., line 16). When a non-structured knowledge field constitutes an inout/out knowledge of a process, the associated method parameter is for technical reasons (related to Java immutable types) passed inside an OutWrapper object (e.g., line 30).

Periodic scheduling of a process is defined via the @DEECoPeriodicScheduling annotation of the process's method, which takes the period expressed in milliseconds in its parameter (line 25). Triggered scheduling is defined via @DEECoTriggered annotation of the method's parameter, change of which should trigger the execution of the process (lines 16-17).

4.1.2 Ensemble

The ensemble definition takes also the form of a Java class. In particular, the class is marked with the @DEECoEnsemble annotation and extends the Ensemble class (Figure 7).

Both the membership predicate and the knowledge exchange are defined as specifically-annotated static methods of this class. While the method representing the membership predicate is annotated by @DEECoEnsembleMembership (line 5), the method representing knowledge exchange is annotated by @DEECoEnsembleKnowledgeExchange (line 19). Note that in the prototype implementation of jDEECo we assume for simplicity knowledge exchange between the coordinator and a single member (applied for each member separately); this is a simplification of the one-to-many knowledge exchange (one coordinator vs. many members) as introduced in Section 3.1.2. Thus, in the Java implementation of the UpdateAvailabilityInformation knowledge exchange we use a timestamp to distinguish current elements of the availabilities

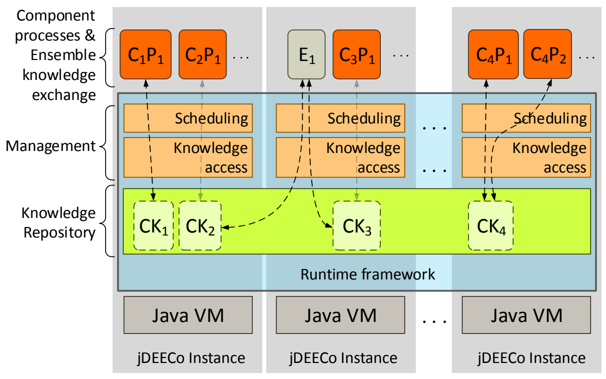


Figure 8: jDEECo runtime framework architecture.

collection (line 27), instead of refreshing the whole collection (Figure 4, line 9).

In contrast to the conceptual description of an ensemble (Section 3.1.2), Java definition of an ensemble does not comprise explicit definition of the member and coordinator interfaces. Instead, these interfaces are defined implicitly as a union of the knowledge fields represented by parameters of the methods representing the membership predicate and knowledge exchange. Since these parameters are annotated in the same way as parameters of component processes, the parameters relevant to the member/coordinator interface are distinguished by identifier prefixes (i.e., identifiers of knowledge of a coordinator/member interface are prefixed with *"coord"*/*"member"*).

Scheduling of the knowledge exchange is defined similarly to component processes. The only difference is that the @DEECoPeriodicScheduling is applied to the whole class defining the ensemble, while the @DEECoTriggered is applied to a particular parameter of the membership method.

4.2 Runtime framework

The jDEECo runtime framework is primarily responsible for scheduling component processes, forming ensembles, and performing knowledge exchange. It also allows for distribution of components.

As illustrated in Figure 8, it is internally composed of the management part and the knowledge repository. The management part is further composed of two modules. One is responsible for scheduling and execution of component processes and knowledge exchange of ensembles. The other is responsible for managing access to the knowledge repository. Exploiting the fact that all modules of the runtime framework implementation are loosely coupled, we are able to introduce implementation variants for each of them. As a result, different variants can be selected in order to reflect specific requirements imposed to the platform.

The role of the knowledge repository is to store the component's knowledge (e.g., CK_1 – knowledge of component C_1 – in Figure 8). Its responsibility is also to provide component processes and knowledge exchange of ensembles with access to this knowledge. In fact, we provide a local and a distributed implementation of the knowledge repository; the former is employed for simulation and verification of the code (Section 4.3) while the latter is used in case the runtime framework needs to run in a distributed setting (i.e., the distribution is achieved at the level of knowledge repository). Specifically, the distributed implementation of the knowledge repository allows each component to run in a different Java virtual machine (as illustrated in Figure 8). The distribution is achieved by employing the JavaSpaces[1] middleware. JavaSpaces is a reification of the LINDA [15] paradigm, which aligns well with the way DEECo represents knowledge. For the time being, jDEECo relies on the ApacheRiver[2] implementation of JavaSpaces.

As to the scheduling module, each component process (e.g., C_1P_1 – process P_1 of component C_1 – in Figure 8) is executed by the runtime framework within a regular Java thread. Thus, threads executing triggered processes are blocked till their triggering condition holds true, while threads executing periodic processes are blocked after completion till the beginning of their next period. Concerning knowledge exchange of ensembles (e.g., E_1 in Figure 8), the scheduling and execution is similar to component processes. In addition, the membership predicate is evaluated before each run of the knowledge exchange, so that it is applied only to valid coordinator-member pairs of components.

Further, to enable dynamic deployment of DEECo-based applications, Java classes with component/ensemble definitions can be provided to the runtime framework both during deployment and runtime.

4.3 Tool support

In addition to providing the runtime framework, jDEECo supports the development of DEECo-based applications via the ASCENS tool workbench (called SDE[3]), featuring modeling and analysis tools for RDS.

Since SDE is based on Eclipse, the integration with jDEECo includes deploying jDEECo as an Eclipse plugin and providing additional Eclipse-specific features. Most importantly, these include the possibility of packaging and deploying DEECo components and ensembles as OSGi [17] bundles. This is complemented by a graphical packaging tool and a discovery mechanism based on OSGi service discovery.

Furthermore, the tool palette is enhanced by the integration of jDEECo and Java PathFinder[4] [18] which supports verification of properties related to knowledge. Currently, we are focusing on verification of reachability properties, encoded via assertions and exceptions in the component/ensemble code. Technically, we perform model-checking on a compound consisting of code of components and ensembles, and of the jDEECo runtime framework. The latter is included to represent the DEECo computational model. To minimize model-checking complexity, we perform the verification on a special configuration of the jDEECo runtime framework (its JPF-optimized variant); in particular, this concerns the local knowledge repository and scheduling module.

5. SOFTWARE ENGINEERING PROCESS INTEGRATION

To build EBCS-based systems (DEECo-based applications in particular) and reason about their properties in a systematic way, a high-level view of the target system is required. Such view should trace the (latent) system architecture, which will naturally comprise a number of DEECo components and ensembles, back to system requirements.

[1] http://river.apache.org/doc/specs/html/js-spec.html
[2] http://river.apache.org
[3] http://sde.pst.ifi.lmu.de/trac/sde/
[4] http://babelfish.arc.nasa.gov/trac/jpf/

To enable that, we have proposed a requirements-driven method for designing EBCS, called *Invariant Refinement Method – IRM* (elaborated in [9][24]). In this section, we augment the description of the DEECo component model and its jDEECo runtime framework implementation with a comprehensive development process based on IRM. In particular, for convenience we first provide a brief summary of IRM and then focus specifically on its integration with traditional Component-Based Development (CBD) process, as well as its strong points w.r.t. system evolution.

5.1 Basic Concepts of IRM

IRM is based on the systematic decomposition and refinement of system specification, ending up with system architecture – components and ensembles. It builds on the idea of iterative refinement of system goals, employed in goal-oriented requirements engineering. Contrary to classic goal-oriented approaches though, like KAOS [27] and Tropos/i* [5], IRM is tailored to the domain of EBCS. In particular, EBCS feature emergent system architectures, which cannot be systematically derived from system requirements using classic approaches [16].

The main goal of IRM is the identification of EBCS concepts of components and ensembles based on system requirements. This subsequently brings correct-by-construction guarantees of compliance with system requirements, and the possibility of automated preparation of EBCS artifacts (component skeletons, ensemble code) in the programming language of choice.

IRM comprises system level design, ensemble level and component level design, followed directly by implementation.

System level. As a starting point of the design process, IRM focuses on the *invariants* to be preserved and the *system constituents (components)* responsible for preserving them. Invariants are descriptive statements of what should hold in the system at every time instant (not only at some point in the future) and reflect the system normalcy, i.e., the property of being within the bounds of normal operation. For example, the *"The availability of relevant PLCSs is kept updated"* invariant expresses that vehicles should keep having up-to-date availability information regarding the PLCSs close to their POIs. A component in IRM is a design construct encapsulating knowledge (its domain-specific data) that is referred from invariants; i.e., the component takes a *role* in the invariants.

After identifying the invariants reflecting the top-level system goals/requirements, the design process continues by their refinement into sets of sub-invariants, forming a tree structure. The invariant refinement has the typical semantics used in software engineering, where the composition of the children exhibits all the behavior expected from the parent and potentially some more. An example of a possible decomposition of our running example is depicted in Figure 10.a.

The iterative refinement process ends when all invariants are directly mappable to DEECo component processes and ensembles. In particular, an invariant needs no further refinement when a) it involves a single component and can be ensured by local manipulation of the component's knowledge (via a component process) – *local invariant* (e.g., (7) in Figure 10.a) – or b) the invariant involves exactly two components and can be ensured by mapping one component's knowledge part(s) to the other (via knowledge exchange of an ensemble) – *exchange invariant* (e.g., (6) in Figure 10.a).

Ensemble level. At this level, ensembles are identified and fully specified. For each exchange invariant, an ensemble is introduced. In particular, the coordinator and member interfaces are directly

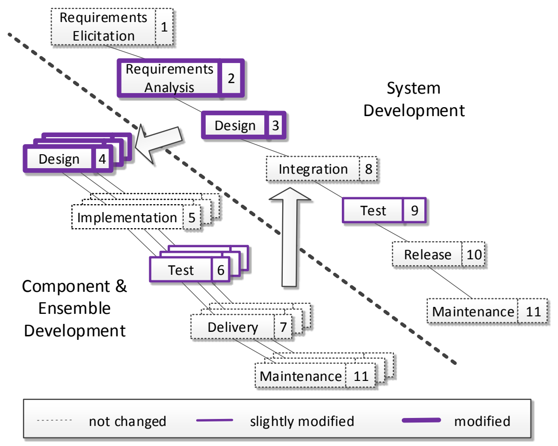


Figure 9: Example of IRM integration into the reference CBD process of [12].

derived from the roles the components take in the respective invariant. The rest of the ensemble definition (membership predicate, knowledge exchange function) needs to be extracted from the invariant manually. For example, the *"The availability of relevant PLCSs is kept updated"* exchange invariant ((6) in Figure 10.a) can be refined into the UpdateAvailabilityInformation ensemble listed in Figure 4.

Component level. At this level, the components are concretized. The component at this level necessarily comprises the knowledge identified at the system level. The component processes are also specified; these are derived from the local invariants the component takes a role in. For example, the Vehicle component from Figure 10.a can be concretized into the Vehicle component of Figure 3, comprising knowledge and processes determined at the system level.

5.2 Integration with CBD Process

Overall, the development process for EBCS as described above, and IRM in particular, introduces specific aspects into the traditional Component-Based Development (CBD) process. Thus, in this section we elaborate on these specifics in the context of general CBD process and provide a concrete example for the waterfall-based CBD process as proposed in [12].

CBD process builds on separation of system development process from component development process [11]. The traditional system development process includes the phases of Requirements, Analysis, Design, Implementation, Test, Release, and Maintenance. The component development process includes phases of Design, Implementation, Test, Delivery, and Maintenance. Several component development processes may be on course simultaneously, making it possible to develop several components at the same time.

By employing IRM in design, we couple component development (exemplified on the reference CBD process of [12] in Figure 9) with ensemble development. To do so, we extend several phases of CBD to accommodate IRM (Table 1). Since the extensions do not rely on any specifics of CBD (they only assume requirements analysis and architectural/system design, traditional parts of development processes in general), we believe that they are applicable to any development process which involves components (e.g., agile variations of CBD).

Req. Analysis [2]	By applying the IRM method, the requirements are captured in terms of invariants and elaborated by iterative refinement.
System Design [3]	The system architecture, in terms of (DEECo) ensembles and components, is identified. The analysis is both structural (which architectural entities should be present in the system) and behavioral (what should be their behavior, e.g., in terms of process & ensemble scheduling). It is important to distinguish between the components' internal and external interfaces. An external interface comprises a part of the knowledge that can be exchanged (read or written) by ensembles. This knowledge must not be violated during implementation, as this would harm the system-wide contractual design. On the contrary, an internal interface comprises a part of the knowledge that must be present in the component, for the purpose of an internal computation.
Comp. Design [4]	Components & ensembles are designed in detail. This step can include elaboration of representation of the knowledge belonging to internal interfaces.
Comp. Testing [6]	Components & ensembles are tested in isolation. The leaf invariants of the IRM tree can serve as a specification for unit testing.
System Testing [9]	System-wide tests are performed. The non-leaf invariants of the IRM tree can serve as a specification for integration testing.

Table 1: IRM injection points into the CBD process.

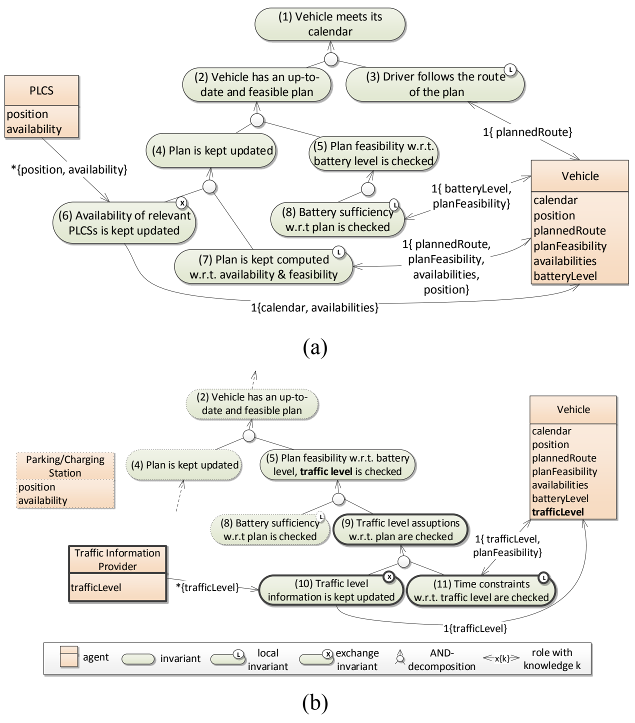
5.3 System Evolution

Since EBCS are inherently open-ended and evolving systems, the aforementioned development process has to accommodate additional requirements that arise after the initial development cycle has been completed. A new requirement can arise when a new or modified functionality is required from the system. IRM provides an easy and effective way to deal with such evolution by introducing new invariants into corresponding branches of the IRM tree.

For illustration, we consider an evolution scenario where the Traffic Information Provider component is added to the system, to represent the traffic monitoring stations scattered around roads. These stations provide information to the vehicles about traffic congestions in their vicinity. Recall that the e-mobility system from the running example has been originally designed and implemented without considering traffic level information (Figure 10.a). In this case, the IRM design captures just the necessity to keep the vehicle's plan updated ((4) in Figure 10.a) and to check whether the current plan remains feasible with respect to measured energy level ((5) in Figure 10.a).

To address the evolution, the IRM tree is modified as follows (Figure 10.b): i) the new component is added, ii) the invariant (5) is modified to account for the traffic level, iii) three new invariants (i.e., (9), (10), (11)) are added. Out of these, one is an exchange invariant (10) and one is a local invariant (11), prescribing the addition of a new ensemble and a new process to the Vehicle component.

To account for such kind of system evolution, the whole development process needs to follow an iterative approach, where, by integrating newly identified requirements, software is incrementally built, tested, and released.



(1) Vehicle meets its calendar
(2) Vehicle has an up-to-date and feasible plan
(3) Driver follows the route of the plan
PLCS
position
availability
1{ plannedRoute}
(4) Plan is kept updated
(5) Plan feasibility w.r.t. battery level is checked
*{position, availability}
Vehicle
calendar
position
plannedRoute
planFeasibility
availabilities
batteryLevel
1{ batteryLevel, planFeasibility}
(6) Availability of relevant PLCSs is kept updated
(8) Battery sufficiency w.r.t plan is checked
1{ plannedRoute, planFeasibility, availabilities, position}
(7) Plan is kept computed w.r.t. availability & feasibility
1{calendar, availabilities}
(a)
(2) Vehicle has an up-to-date and feasible plan
Parking/Charging Station
position
availability
(4) Plan is kept updated
(5) Plan feasibility w.r.t. battery level, traffic level is checked
Vehicle
calendar
position
plannedRoute
planFeasibility
availabilities
batteryLevel
trafficLevel
(8) Battery sufficiency w.r.t plan is checked
(9) Traffic level assuptions w.r.t. plan are checked
1{ trafficLevel, planFeasibility}
Traffic Information Provider
trafficLevel
*{trafficLevel}
(10) Traffic level information is kept updated
(11) Time constraints w.r.t. traffic level are checked
1{trafficLevel}
agent
invariant
local invariant
exchange invariant
AND-decomposition
x{k}
role with knowledge k
(b)

Figure 10: Capturing system evolution in IRM.

6. EXPERIENCE

We have evaluated the DEECo approach (together with IRM) by developing a prototype of the e-mobility case study within the ASCENS project. As this case study has been conceived in cooperation with Volkswagen, the detailed designs and implementation are proprietary. For a concise description of the case study we refer the reader to [36]. Along with the case study, we have also implemented a number of example applications and a tutorial, which are all available at the jDEECo GitHub site [13].

Our experience shows that DEECo concepts well combine the encapsulation and modularity brought by components with the needs of autonomic behavior and highly dynamic architecture. IRM process well complements the DEECo concepts in providing an overall system-level view that can be easily translated to components and ensembles. The mapping to Java (by jDEECo) proved to be relatively straightforward.

Our experience also indicated that although there is a strong conceptual difference between a component and an ensemble (in the sense that a component is state-full while an ensemble is stateless), the developers of the case-study had problems with differentiating between responsibilities of a component process and knowledge exchange. In particular, they incorrectly tended to reduce autonomy of components by pushing some of their functionality to ensembles (by employing complex knowledge transformations in the knowledge exchange). As a remedy, we adopted the following rule as a design guideline: The knowledge exchange should be ideally 1:1 knowledge assignment; complex knowledge transformations may be employed only in well-justified cases (typically when integrating third-party components).

Finally, our experiments with verification of jDEECo applications via JPF (performed on the example applications) indicate that the relatively strict DEECo computational model can be effectively exploited for increasing the performance of explicit model checking.

7. RELATED WORK

Since EBCS are a relatively new class of systems, we are currently not aware of any other approach that would be directly related to IRM and DEECo. However, as EBCS is a software engineering concept for developing Resilient Distributed Systems (RDS), in this section we survey approaches that deal with specific aspects of RDS.

At the computational level, control engineering methodologies have been identified as a promising solution to implement self-adaptive software systems [10] in a variety of application domains and with different performance requirements and control objectives [33]. In the domain of distributed systems, decentralized solutions based on feedback loops, ranging from cloud performance management [41] to embedded real-time systems [40], have been proposed to keep the system in the required steady state, while avoiding scalability issues and single points of failure. EBCS employ similar idea of cyclic execution of component processes and ensembles to maintain the operational normalcy of the system. At the architectural level, attempts have been made to instantiate the generic MAPE-K loop [23] to feature adaptation at a larger scale. Self-managing architectures [26], component-based approaches [3][34], and solutions that apply architectural models at runtime [29] are examples of this. The common denominator of these approaches is that they rely on explicit bindings among the system components, which get re-organized in response to runtime stimuli. EBCS, on the other hand, do not consider explicit architecture, but let the architecture "emerge" during runtime, fitting better the dynamic, constantly–changing system landscapes.

Agent-oriented approaches provide useful notions (e.g., goals, plans), models (e.g., Belief-Desire-Intention [35]) and algorithms (e.g., DCOPs [21]) for reasoning in complex dynamic systems. In a distributed setting, multi-agent analysis is based on the conceptual autonomy and social ability of the parts constituting the system. A problem is that current agent implementation platforms [4] and methodologies [5] rely on guaranteed communication and explicit bindings among the agents, which typically take the form of messaging. In this view, EBCS/DEECo stands as an agent engineering platform, which handles the communication in an implicit and automatic way, making it possible for agents to operate in opportunistic environments where no guarantees are available.

The concept of service-component ensembles has been recently proposed in order to allow for communication over unreliable communication channels and at massive scale [20]. Ensembles rely on attribute-based communication [14] to model a best-effort, dynamic coordination of components. An attempt to formally define this concept can be found in [19].

At the requirements phase, well-established methods and models exist for capturing and analyzing early requirements in terms of goals delegated to system agents. However, these models either do not map effectively to the later development phases [27], or do not support mapping to emergent architectures [5], which are typical in EBCS. Recent attempts in the area of EBCS have centered around a model termed Statement of the Affairs (SOTA), which provides the means to capture and analyze the early requirements of different component cooperation schemes, along with the architectural patterns that satisfy them by construction [1]. IRM stands as the intermediate method which guides the transition from early (high-level) requirements to system architecture in terms of components and ensembles.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have focused on Resilient Distributed Systems (RDS). We have argued that classic component-based approaches in design do not scale well in the area of RDS – mainly because RDS exhibit very high degree of dynamicity, adaptivity, and autonomy.

For component-based development of RDS, we have introduced EBCS (Ensemble-Based Component Systems), a new class of component-based systems, which combine concepts from agent-oriented, ensemble-oriented and control systems. In particular, we have presented an instance of EBCS – the DEECo component model and its framework.

Overall, DEECo provides a comprehensive software engineering solution comprising (i) component and ensemble paradigms with well-defined formal semantics, (ii) mapping to Java, (iii) distributed Java-based runtime framework (jDEECo), (iv) integration with analysis tools (SDE, JPF), (v) design method (IRM) for deriving components and ensembles from high-level requirements, and (vi) integration of the design method to traditional component-based development processes. We have successfully evaluated DEECo along with IRM on the e-mobility case-study of the ASCENS project.

The experience with DEECo (and consequently EBCS) puts forward several research directions. In particular we would like to evaluate the robustness of DEECo in environments with highly unreliable communication and heterogeneous network infrastructure (e.g., MANETs [28]). Although this will most likely require employing some communication middleware for such networks (e.g., EgoSpaces [22]) at the implementation level, it is well aligned with the general DEECo computational model. Also, we are currently investigating the possibility of using formalized IRM invariants as the basis for monitoring the correctness and performance of a DEECo-based system and for guiding component adaptations. Furthermore, we intend to develop a metamodel of DEECo and employ model-driven-engineering techniques for elaborating the jDEECo implementation.

9. ACKNOWLEDGMENTS

The authors would like to thank Pavel Parizek and Pavel Jancik for their input concerning JPF. This work was partially supported by the EU project ASCENS 257414 and the Grant Agency of the Czech Republic project P202/11/0312. The work was also partially supported by Charles University institutional funding SVV-2013-267312.

10. REFERENCES

[1] D. B. Abeywickrama, N. Bicocchi, and F. Zambonelli. SOTA: Towards a General Model for Self-Adaptive Systems. In *Proc. of WETICE '12*, 2012.

[2] R. Al Ali, T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil. *DEECo computational model – I*. Technical Report no. D3S-TR-2013-01. D3S, Charles University in Prague. Available at: http://d3s.mff.cuni.cz-/publications, 2013.

[3] L. Baresi, S. Guinea, and G. Tamburrelli. Towards decentralized self-adaptive component-based systems. In *Proc. of SEAMS '08*, 2008.

[4] F. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley, 2007.

[5] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*. 8, 3, 2004.

[6] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J. Stefani. The Fractal component model and its support in Java. *Software: Practice & Experience*. 36, 2006.

[7] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0 : Balancing Advanced Features in a Hierarchical Component Model. In *Proc. of SERA '06*, 2006.

[8] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil. Autonomous components in dynamic environments. *Awareness Magazine*. Online: http://www.awareness-mag.eu, 2012

[9] T. Bures, I. Gerostathopoulos, V. Horky, J. Keznikl, J. Kofron, M. Loreti, and F. Plasil. *Language Extensions for Implementation-Level Conformance Checking*. ASCENS Deliverable 1.5. Available at: http://www.ascens-ist.eu/deliverables, 2012.

[10] B. Cheng et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. *Software Engineering for Self-Adaptive Systems*. Springer–Verlag, 2009.

[11] I. Crnkovic. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[12] I. Crnkovic, M. Chaudron, and S. Larsson. Component-based development process and component lifecycle. *Software Engineering Advances*, 44, 2006.

[13] D3S, Charles University in Prague. *jDEECo website*. Accessed April 17, 2013. https://github.com/d3scomp/JDEECo, 2013.

[14] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese. A Language-based Approach to Autonomic Computing. In *Proc. of FMCO '11*, 2012.

[15] D. Gelernter. Generative communication in Linda. *Toplas*. 7, 1, 1985.

[16] I. Gerostathopoulos, T. Bures, and P. Hnetynka. Position Paper: Towards a Requirements-Driven Design of Ensemble-Based Component Systems. In *Proc. of HotTopiCS Workshop, ICPE '13*, 2013.

[17] R. Hall, K. Pauls, S. McCulloch, and D. Savage. *OSGi in Action: Creating Modular Applications in Java*. Manning Pubs Co Series. Manning Publications, 2011.

[18] K. Havelund, and T. Pressburger. Model Checking Java Programs Using Java PathFinder. *Software Tools for Technology Trasfer*. 2, 4, 2000.

[19] M. Holz, and M. Wirsing. Towards a System Model for Ensembles. *Formal modeling*. 2012.

[20] M. Holzl, A. Rauschmayer, and M. Wirsing. Engineering of software-intensive systems: State of the art and research challenges. In *Software-Intensive Systems and New Computing Paradigms*. Ser. LNCS, Springer Berlin, Heidelberg, vol. 5380, 2008.

[21] M. Jain, M. Taylor, M. Tambe, and M. Yokoo. DCOPs meet the real world: Exploring unknown reward matrices with applications to mobile sensor networks. In *Proc. of IJCAI '09*, 2009.

[22] C. Julien, and G.-C. Roman. EgoSpaces: Facilitating Rapid Development of Context-Aware Mobile Applications. *IEEE Transactions on Software Engineering,* 32, 5, 2006.

[23] J. Kephart, and D. Chess. The Vision of Autonomic Computing. *Computer*. 36, 1, 2003.

[24] J. Keznikl, T. Bures, F. Plasil, I. Gerostathopoulos, P. Hnetynka, and N. Hoch. Design of Ensemble-Based Component Systems by Invariant Refinement. In *Proc. of CBSE 2013*, ACM, 2013.

[25] J. Keznikl, T. Bures, F. Plasil, and M. Kit. Towards Dependable Emergent Ensembles of Components: The DEECo Component Model. In *Proc. of WICSA/ECSA 2012*, IEEE CS, 2012.

[26] J. Kramer, and J. Magee. Self-managed systems: an architectural challenge. In *Proc. of FOSE '07*, 2007.

[27] A. Lamsweerde. Requirements engineering: from craft to discipline. In *Proc. of SIGSOFT '08/FSE-16*, 2008.

[28] M. Mauve, A. Widmer and H. Hartenstein. A Survey on Position-Based Routing in Mobile Ad Hoc Networks. *IEEE Network*, 15, 6, 2001.

[29] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models at Runtime to Support Dynamic Adaptation. *IEEE Computer*. 42, 10, 2009.

[30] OMG. *Unified Modeling Language 2.0: Superstructure*. Available online: http://www.omg.org/spec/UML/2.0/, 2005.

[31] OMG. *CORBA Component Model Specification v4.0*. Available online: http://www.omg.org/spec/CCM/4.0/, 2006.

[32] OSGi Alliance. *OSGi service platform core specification, release 4*. Available online: http://www.osgi.org/Spec-ifications/HomePage, 2012.

[33] T. Patikirikorala, A. Coman, H. Jun, and W. Liuping . A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *Proc. of SEAMS '12,* 2012.

[34] C. Peper, and D. Schneider. Component engineering for adaptive ad-hoc systems. In *Proc. of SEAMS '08*, 2008.

[35] A. Rao, and M.P. Georgeff. BDI agents: From theory to practice. In *Proc. of ICMAS '95*, 1995.

[36] N. Serbedzija et al. *Ensemble Model Syntheses with Robot, Cloud Computing and e-Mobility*. ASCENS Deliverable 7.2. Available at: http://www.ascens-ist.eu/deliverables, 2012.

[37] N. Serbedzija, S. Reiter, M. Ahrens, J. Velasco, C. Pinciroli, N. Hoch, and B.Werther. *Requirement Specification and Scenario Description of the ASCENS Case Studies*. ASCENS Deliverable 7.1. Available at: http://www.ascens-ist.eu/deliverables, 2011.

[38] M. Shaw, and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

[39] Y. Shoham, and K. Leyton-Brown. *Multiagent Systems: Algorithmic, GameTheoretic, and Logical Foundations,* Cambridge University Press, 2008.

[40] J. A. Stankovic, T. He, T. Abdelzaher, M. Marley, G. Tao, S. Son, and C. Lu. Feedback control scheduling in distributed real-time systems. In *Proc. of RTSS '01*, 2002.

[41] R. Wang, and N. Kandasamy. A distributed control framework for performance management of virtualized computing environments. In *Proc. of ICAC '10*, 2009.

3.3 Design of Ensemble-Based Component Systems by Invariant Refinement

Jaroslav Keznikl,
Tomáš Bureš,
František Plášil,
Ilias Gerostathopoulos,
Petr Hnětynka,
Nicklas Hoch

In proceedings of the **16th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE '13)**.

Awarded with the Distinguished Paper Award.

Published by ACM,
pages 91-100,
ISBN 978-1-4503-2122-8,
June 2013.

The original version is available electronically from the publisher's site at http://dx.doi.org/10.1145/2465449.2465457.

Summary of the Paper

The goal of this paper, published as [KBP+13], is to address the challenge of systematic architecture design of dependable RDS (i.e., C4 in Section 1.3), DEECo-based systems in particular. (Note that DEECo is presented under the umbrella of Ensemble-Based Component Systems, similar to Section 3.2.) As discussed in [GBH13], this challenge stems from the fact that, although effectively addressing the requirements of RDS at the level of component model, the concept of ensemble cannot be properly exploited at design time when using the traditional software engineering methods. Due to the dynamic and stateless nature of the ensemble concept, it is especially problematic to systematically determine a proper DEECo-based architecture (i.e., components, component processes, and ensembles) of a system, given the system's overall goals and requirements. This in turn prevents validation and verification of design decisions. The root of this problem is the conceptual gap between the high-level system goals and the architecture abstractions of DEECo.

While pursuing the research goal **G2**, the paper responds to this challenge by introducing the *Invariant Refinement Metho*d (IRM). IRM is a formally-grounded design method, building on goal-based requirements elaboration [RBAF10, VL01], that embraces the specifics of the architecture abstractions of DEECo and exploits them for systematically driving the design from high-level requirements to a DEECO-based architecture in a way that the compliance of design decisions with the overall system goals and requirements is explicitly captured. Being formally grounded, the design process of IRM allows for design validation and verification.

IRM exploits the fact that the objective of a component process/an ensemble in DEECo is (adopting the perspective of real-time software control systems) to maintain an *operational normalcy* of the corresponding component/group of components. Here, operational normalcy expresses the property of being within certain limits that define the range of normal functioning of the component/group of components. As a result, a component process, as well as an ensemble, can be described in terms of the particular operational normalcy it maintains. In IRM, this is expressed formally via *invariants*, which describe how the normalcy projects onto the knowledge evolution of the corresponding component/group of components. Assuming that the high-level system goals can be also described via invariants, the objective of IRM is to start with the invariants corresponding to the overall system goals and, by employing systematic *refinement*, end up by determining the invariants reflecting detailed design of the particular system constituents – components and ensembles. Reasoning along the lines of what needs to be maintained (expressed via invariants) as opposed to what needs to be performed (actions) or what should hold in the future (goals) allows expressing the relation of a component to its environment and itself. This is particularly valuable for the design of autonomous adaptive RDS that continuously interact with their environment.

The refinement in IRM takes the form of gradual decomposition (i.e., structural elaboration) of a higher-level invariant into a conjunction of lower-level sub-invariants so that, formally, the conjunction of the sub-invariants implies the parent invariant. This complies with the traditional interpretation of refinement, where the composition of the refinement products (e.g., sub-components) exhibits the behavior expected from the refined concept (e.g., a composite component) and potentially more. This way, IRM provides an appropriate level of abstraction and separation of concerns. Since there can be many possible refinements, the decomposition step may involve a design decision. The rule of thumb is that refinement is finalized when each leaf invariant of the refinement tree represents either an *assumption* about the environment or captures normalcy to be maintained by a single component process/ensemble, i.e., it is a *process/exchange invariant*. Each process invariant is subsequently refined into a component process and each exchange invariant into an ensemble.

Performing the decomposition step is a relatively complex task, since it has to bridge different levels of abstraction (i.e., from high-level requirements to low-level architecture aspects). This is especially true when requiring formal compliance with the refinement semantics. To this end, the paper presents five formally defined patterns of invariants capturing the specifics of invariants at different levels of abstraction. Based on these patterns, and backed with a formal framework, the paper provides guidelines for invariant decomposition at the same level or bridging adjacent levels of abstraction.

Similar to [BGH+13] (Section 3.2), the contributions of the paper are illustrated and evaluated on the electrical vehicle navigation case study featured by the ASCENS project [SMP+12, SRA+11].

Comments on Authorship

Although the main idea of the paper is of equal authorship, I contributed to this paper by elaborating and formalizing the idea, as well as addressing the technical details. I also personally contributed with the idea and formalization of the invariant patterns. Finally, with the indispensable support of the other authors, I authored a majority of the text.

Design of Ensemble-Based Component Systems by Invariant Refinement

Jaroslav Keznikl[1,2]
keznikl@d3s.mff.cuni.cz

Tomas Bures[1,2]
bures@d3s.mff.cuni.cz

Frantisek Plasil[1]
plasil@d3s.mff.cuni.cz

Ilias Gerostathopoulos[1]
iliasg@d3s.mff.cuni.cz

Petr Hnetynka[1]
hnetynka@d3s.mff.cuni.cz

Nicklas Hoch[3]
nicklas.hoch@volkswagen.de

[1]Charles University in Prague
Faculty of Mathematics and Physics
Prague, Czech Republic

[2]Institute of Computer Science
Academy of Sciences
of the Czech Republic
Prague, Czech Republic

[3]Corporate Research Group
Volkswagen AG
Wolfsburg, Germany

ABSTRACT

The challenge of developing dynamically-evolving resilient distributed systems that are composed of autonomous components has been partially addressed by introducing the concept of component ensembles. Nevertheless, systematic design of complex ensemble-based systems is still a pressing issue. This stems from the fact that contemporary design methods do not scale in terms of the number and complexity of ensembles and components, and do not efficiently cope with the dynamism involved. To address this issue, we present a novel method – Invariant Refinement Method (IRM) – for designing ensemble-based component systems by building on goal-based requirements elaboration, while integrating component architecture design and software control system design.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems – d*istributed applications*; C.3 [**Special-purpose and Application-based Systems**]: r*eal-time and embedded systems*; D.2.2 [**Software Engineering**]: Design Tools and Techniques – *miscellaneous*; D.2.11 [**Software Engineering**]: Software Architectures – *patterns.*

Keywords

Component; ensemble; refinement; requirements engineering; system design

1. INTRODUCTION

Addressing the challenge of developing large-scale distributed autonomic and adaptive systems [26], the EU FP-7 project ASCENS [15] strives for modeling and designing such systems of service components and service component ensembles. For large-scale adaptive systems, the ASCENS case studies indicate the need to deal with large amounts of distributed information both highly dynamically and intelligently, while ensuring resilience to changes in the environment. This has been partially targeted by the work on resilient distributed systems (RDS) based on *ensembles* [15] of autonomous adaptive [16] components. In this context, an ensemble is seen as a dynamically formed group of autonomous components which encapsulates knowledge, interaction, and goals specific to the group.

The ASCENS project employs three case studies from different domains, of which we target the e-mobility case study within the scope of this paper. This case study aims at resource optimization, such as travel time, energy consumption, and parking lot and charging station usage of electric-powered vehicles. Its objective is to coordinate planning of journeys in compliance with parking and charging strategies in the highly-dynamic, complex, and heterogeneous traffic environment, where information is distributed.

Currently, widely accepted semantics of the ensemble concept is still an open issue. In [5][19], we have contributed to this by introducing the concept of *Ensemble-Based Component Systems* (EBCS) and specifically the DEECo component model (Dependable Emergent Ensembles of Components), our contribution to the EBCS family. Although the concept of ensemble in EBCS effectively addresses the distribution and dynamism of RDS at a middleware level, the design of complex, ensemble-based systems remains a significant challenge. Our early experiments indicate that traditional software engineering methods cannot be directly employed [13], since they cannot cope with the dynamism involved and do not cover all the required design steps. Specifically, it appears that the design of ensemble-based systems requires a synergy of goal-oriented requirements refinement, architecture design, and (real-time) process scheduling. In response to this problem, this paper proposes a novel method – Invariant Refinement Method (IRM) – for systematical derivation of an EBCS-based RDS architecture from high-level requirements. In particular, IRM builds on gradual refinement of invariants that are employed as a concept for reflecting both requirements and architectural elements.

The rest of this paper is structured as follows: Section 2 explains the specifics of EBCS in the context of the e-mobility case study in DEECo. Section 3 elaborates on the lessons learned from the case study and articulates the problem statement. Section 4 presents an overall description of IRM, while Section 5 elaborates on guidelines for refinement by presenting invariant patterns. The evaluation and discussion is provided in Section 6 and related work in Section 7. Section 8 concludes the paper and identifies future research directions.

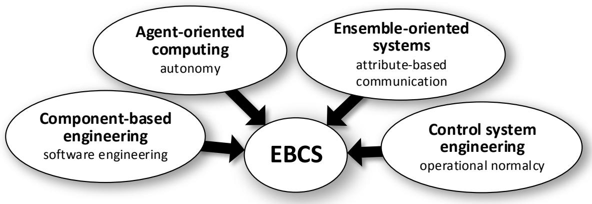

Agent-oriented computing
autonomy
Ensemble-oriented systems
attribute-based communication
Component-based engineering
software engineering
EBCS
Control system engineering
operational normalcy

Figure 1: Context of Ensemble-Based Component Systems (EBCS).

2. ENSEMBLE-BASED COMPONENT SYSTEMS: A CASE STUDY

To illustrate the challenges in RDS development, we exploit the e-mobility case study mentioned in Section 1. Electric vehicles (*e-vehicles*) compete for e-mobility resources, such as parking lots and charging stations (*infrastructure*) in order to achieve optimal journeys with respect to the drivers' daily activities (*calendars*). A calendar consists of a set of *points of interest* (POIs), together with timing constraints specifying the expected POI arrival and departure times. For brevity, we assume that each driver is bound to his/her own private vehicle and that parking lots are the only infrastructure entities. An e-vehicle uses a planner in order to create its individual journey *plan*, stemming from the driver's calendar and including parking/charging periods when necessary. The system is fully decentralized – every e-vehicle plans and executes its route individually.

Having outlined the application domain of EBCS, in the rest of this section we first elaborate on the context of EBCS and then illustrate the basic concepts on an example from the case study.

2.1 From Agent and Control-based Systems to Ensemble-Based Component Systems

In principle, EBCS [5] combine the advantages of component-based software engineering [9][10], ensemble-oriented systems [14][15], agent-based computing [18][24], and (soft) real-time embedded software control systems [7][25] in highly dynamic, open-ended environments that lack reliable communication channels (Figure 1).

Exploitation of the concepts from agent-oriented computing allows for composing systems from a number of autonomous entities, so that the overall behavior of the system is an emergent result of behaviors of the entities. In particular, the autonomous entities are designed to operate only with a partial view of the whole system; i.e., BDI model [21] where agents maintain a *belief* about the rest of the system to guide their autonomous decisions.

A disadvantage of the agent-oriented computing concepts at the software-engineering level is its strong dependence on reliable communication channels (as, e.g., in the case of JADE platform [3]), which is, however, not achievable in the target application domain due to the extreme dynamism. Instead, EBCS rely on the concept of attribute-based communication [12] (i.e., the target of communication is determined according to the values of attributes rather than by a direct identifier), which models the communication as best effort and localized to dynamically changing groups – ensembles – of components.

The EBCS communication model however implies that the components' belief is essentially always outdated. To efficiently cope with outdated belief, EBCS employ concepts of (soft) real-time software control systems, which achieve robustness by

```
interface AvailabilityAggregator:
    calendar, availabilityList

interface AvailabilityAwareParkingLot:
    position, availability

component Vehicle0123 features AvailabilityAggregator, ... :
    knowledge:
        calendar, availabilityList, plan, planFeasibility, ...
    process computePlan(in calendar, in availabilityList, out plan):
        function:
            plan ← JourneyPlanner.computePlan(
                            calendar, availabilityList, planFeasibility)
        scheduling: triggered( changed(planFeasibility) ∨ changed(availabilityList) )
    ...

component ParkingLot01  features AvailabilityAwareParkingLot, ... :
    knowledge:
        position, availability, ...
    process observeAvailability(out availability):
        function:
            availability← Sensors.getCurrentAvailability()
        scheduling: periodic( 2000ms  )
    ...

ensemble UpdateAvailabilityInformation:
    coordinator: AvailabilityAggregator
    member: AvailabilityAwareParkingLot
    membership:
        ∃ poi ∈ coordinator.calendar:
            distance(member.position, poi.position) ≤ TRESHOLD
    knowledge exchange:
        coordinator.availabilityList ← members.reduce(member.availability)
    scheduling: periodic( 5000ms )
```

Figure 2: Example of a DEECo component and ensemble definition in a DSL.

adequate scheduling of periodic tasks recurrently maintaining the *operational normalcy* of the system. Here, operational normalcy expresses the property of being within certain limits that define the range of normal functioning of the system. The required level of robustness is achieved by adjusting the periods of the tasks.

As extreme dynamism is involved, components should be also capable of continuous self-adaptation, following the concept of feedback loops [17]. An ensemble-based system can be thus understood as a dynamic system of conditionally interacting feedback loops.

In this context, components in EBCS are perceived as software-engineering means for implementing resilient agents that deal with ensemble-oriented, best-effort communication and outdated belief.

2.2 Illustration of the Concepts on the Case Study

The case study has been implemented in our DEECo component model – an instance of EBCS. Here, a component comprises *knowledge* (i.e., the data of the component), exposed via a set of *interfaces*, and *processes*, each of them being essentially a thread operating upon the knowledge of the component. Figure 2 illustrates several artifacts we have developed for the case study. In particular, it shows a specification of the Vehicle0123 component, featuring the AvailabilityAggregator interface and the computePlan process. The latter is responsible for the computation of the vehicle's plan, which is based on the vehicle's calendar (calendar) and the availability information of the relevant parking lots (availabilityList) and is executed whenever one of these inputs changes.

For the purpose of separation of concerns and effective handling of dynamism and communication errors, DEECo introduces *ensemble,* a first-class concept, encapsulating dynamic grouping of components and the interaction within the group. In an ensemble a component plays the role of the ensemble's coordinator or one of the members. This is determined dynamically (the task of the runtime framework) according to the *membership* condition specified upon the interfaces expected for the coordinator and members. Specifically, the membership condition determines which components form the coordinator-member pairs of an ensemble. The separation of concerns is brought to such extent, that individual components are not capable of explicit communication with other components. Instead, the interaction among the components forming an ensemble takes the form of *knowledge exchange*, carried out implicitly (by the runtime framework). For example, Figure 2 shows a specification of the UpdateAvailabilityInformation ensemble, an instance of which is to be created for every coordinator, i.e., every component that features the interface AvailabilityAggregator (such as the component Vehicle0123). The members of such an ensemble are all the components featuring AvailabilityAwareParkingLot that are in the proximity (TRESHOLD) to one of the POIs of the coordinating e-vehicle. This effectively includes all the parking lots that are relevant to journey planning of the coordinating e-vehicle. The knowledge exchange, scheduled periodically every 5000ms, ensures that the coordinating e-vehicle obtains the current availability information of all the member parking lots. This periodicity guarantees that the "belief" of the e-vehicle about the availability of parking lot components is current enough.

In summary, a component operates only upon its own local knowledge, which is implicitly updated via knowledge exchange whenever the component is part of an ensemble (technically this is handled by the underlying runtime framework).

3. PROBLEM STATEMENT

The lesson from implementing the case study is that it is problematic to determine a proper EBCS architecture (i.e., components, component processes and ensembles) of the system from the overall goals and requirements. This gets more difficult when we take into account the extent to which knowledge can become outdated (due to delays in knowledge exchange and parallel execution of component processes) and its impact on the overall system behavior.

This problem stems from the conceptual gap between the high-level system goals and relatively low-level architectural concepts of EBCS. A broad, high-level view of the goals is critical when reasoning about global properties of a complex (distributed) system as a whole; e.g., stability-related properties including robustness, adaptability, non-functional properties such as tradeoff between communication overhead and outdated knowledge, etc. Focus on the low-level concepts is equally important for a detailed design and implementation of components and ensembles.

Overall, the key objective of both the component process and ensemble concepts is to maintain a form of operational normalcy of the component/group of components. Therefore, they can be described declaratively in terms of the particular operational normalcy they maintain. In addition, we assume that the high-level system goals can be also described declaratively. Thus, both high-level requirements and low-level architectural concepts can be reflected in the same declarative manner.

Hence, the key challenge we address in this paper is to guide the EBCS design process transparently from high level goals to low-level concepts of system architecture in such a way that the compliance of design decisions with the overall system goals and requirements is explicitly captured and (if possible) formally verified. As a result, tracing a low-level design decision back to its rationale in the system goals and requirements would allow for design validation and verification.

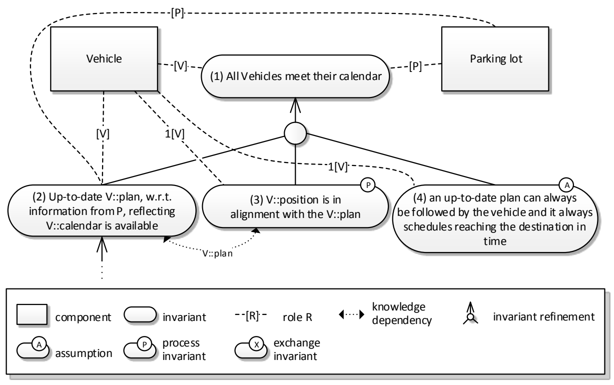

[P]
Vehicle
[V]
(1) All Vehicles meet their calendar
[P]
Parking lot
[V]
1[V]
1[V]
P
A
(2) Up-to-date V::plan, w.r.t. information from P, reflecting V::calendar is available
(3) V::position is in alignment with the V::plan
(4) an up-to-date plan can always be followed by the vehicle and it always schedules reaching the destination in time
V::plan
component
invariant
[R]
role R
knowledge dependency
invariant refinement
A
assumption
P
process invariant
X
exchange invariant

Figure 3: Top-level design of the case study.

4. DESIGNING ENSEMBLES VIA INVARIANT REFINEMENT

To address this challenge, we propose IRM (Invariant Refinement Method) – a novel design method specifically focused on EBCS. Building on goal-based requirements elaboration [22], IRM is based on systematic, gradual refinement (i.e., elaboration) of *invariants* that reflect goals and requirements of the system-to-be [1]. In this context, we are concerned with goals and requirements from the global perspective of the system, rather than the perspective of the individual components and ensembles.

In principle, the invariants describe a desired state of the system-to-be at every time instant; i.e., describe the operational normalcy of the system-to-be, essential for its continuous operation. For example, the main goal of the case study is expressed by the invariant (1): "All Vehicles meet their calendar" (Figure 3).

The objective of IRM is to start the refinement with the overall system goal and end up by determining the invariants reflecting detailed design of the particular system constituents – components, component processes, and ensembles.

4.1 Invariants and Assumptions

A key concept of system design is *component*, i.e., a participant of the system-to-be (e.g., Vehicle and Parking lot in Figure 3). Each component comprises specific knowledge, i.e., its domain-specific data (in Figure 3 left out for brevity). The valuation of components' knowledge evolves in time as a result of their autonomous behavior (i.e., execution of the associated component processes) and knowledge exchange. In principle, an *invariant* is a condition on the knowledge valuation of a set of components that captures the operational normalcy to be maintained by the system-to-be (i.e., that should be preserved as knowledge valuation evolves in time). If a component's knowledge is referenced by an invariant, we say the component takes a *role* in the invariant (e.g., in the invariant (1) from Figure 3 the component Vehicle takes the role V, while Parking lot takes the role P).

As a special case, component knowledge may reflect information about the environment. Consequently, an invariant may represent an *assumption* about the environment, i.e., a condition that is expected to hold during knowledge evolution and thus is not

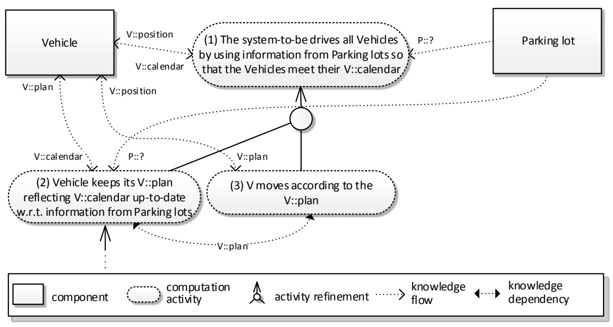

Vehicle
V::position
V::calendar
(1) The system-to-be drives all Vehicles by using information from Parking lots so that the Vehicles meet their V::calendar
P::?
Parking lot
V::plan
V::position
V::calendar
P::?
V::plan
(2) Vehicle keeps its V::plan reflecting V::calendar up-to-date w.r.t. information from Parking lots
(3) V moves according to the V::plan
V::plan
component
computation activity
activity refinement
knowledge flow
knowledge dependency

Figure 4: Dual, computation-activity-based view on the top-level design of the case study from Figure 3.

intended to be maintained explicitly by the system-to-be (in figures marked by A; e.g., (4) in Figure 3).

4.2 Invariants vs. Computation Activities

The underlying idea of IRM is that each invariant which is not an assumption is essentially associated with a *computation activity* – an abstract computation producing *output knowledge* given a particular *input knowledge*. In fact, the computation activity provides a dual view on the invariant – while the invariant reflects an operational normalcy, the computation activity represents means for maintaining it. For example, Figure 4 provides the dual view on the invariants in Figure 3. The invariants thus express the relation between the input and output knowledge of the computation activity. A component process, as well as ensemble knowledge exchange, is a specific form of computation activity.

This dual view gives the convenient option to refer to invariants for the purpose of logic-based reasoning on system-to-be properties and to refer to computation activities when low-level implementation aspects are of concern.

As an aside, we will refer to the relation between component knowledge and input/output knowledge of a computation activity as *knowledge flow*. For example, Figure 4 shows the knowledge flow between Vehicle and the computation activity associated with (3) from Figure 3 (with V::plan, resp., V::position as its input, resp., output knowledge).

The activities associated with high-level system invariants (goals) are abstract, representing the system implementation at a high level of abstraction. For such an abstract computation activity, the input knowledge constitutes the part of the components' knowledge that is out of control of the system-to-be, while the output knowledge is fully in its control. For example, as shown in Figure 4, the input knowledge of the computation activity associated with (1) from Figure 3 comprises V::calendar and potentially some knowledge of parking lots (since it is not yet clear at this level of abstraction, it is denoted by P::?), while its output knowledge comprises V::position.

Thus, in the dual perspective of computation activities, the goal of IRM is to refine such abstract activities into the very concrete component processes and knowledge exchange.

4.3 Invariant Refinement

The core of IRM is a systematic, gradual *refinement* of a higher-level invariant by means of its decomposition (i.e., structural elaboration) into a conjunction of lower-level sub-invariants. Formally, decomposition of a parent invariant I_p into a conjunction of sub-invariants $I_{s1}, \dots, I_{sn}$ is a refinement if the conjunction of the sub-invariants entails the parent invariant, i.e., if it holds that:

1. $I_{s1} \wedge \dots \wedge I_{sn} \Rightarrow I_p$ (entailment)
2. $I_{s1} \wedge \dots \wedge I_{sn} \nRightarrow false$ (consistency)

This definition complies with the traditional interpretation of refinement, where the composition of the children exhibits all the behavior expected from the parent and (potentially) some more.

The refinement is applied recursively, starting with high-level invariants reflecting the overall system goals and involving a number of components and ending with low-level ones involving a single component or an ensemble of components. Note that since a decomposition step may involve a design decision, it is critical to ensure that this decision complies with the entailment and consistency conditions.

During refinement, only the components that take a role in the parent invariant may also take a role in the sub-invariants. Nevertheless, as a part of the design decision, new knowledge can be added into the components taking a role in the sub-invariants (e.g., V::planFeasibility in Figure 5).

In Figure 3, the design decision is to refine the invariant (1) into a conjunction of three sub-invariants: (2) – having an up-to-date plan, (3) – keeping the vehicle's position in alignment with the plan, and (4) – an assumption that an up-to-date plan can always be followed by the vehicle (i.e., the environment dynamics – traffic, parking availability, etc. – will never prevent the car from following an up-to-date plan) and that it always schedules reaching the destination in time.

The sub-invariants can exhibit *knowledge dependency* due to references to the same knowledge of a specific component. For example, in Figure 3 there is a knowledge dependency between (2) and (3) due to references to V::plan.

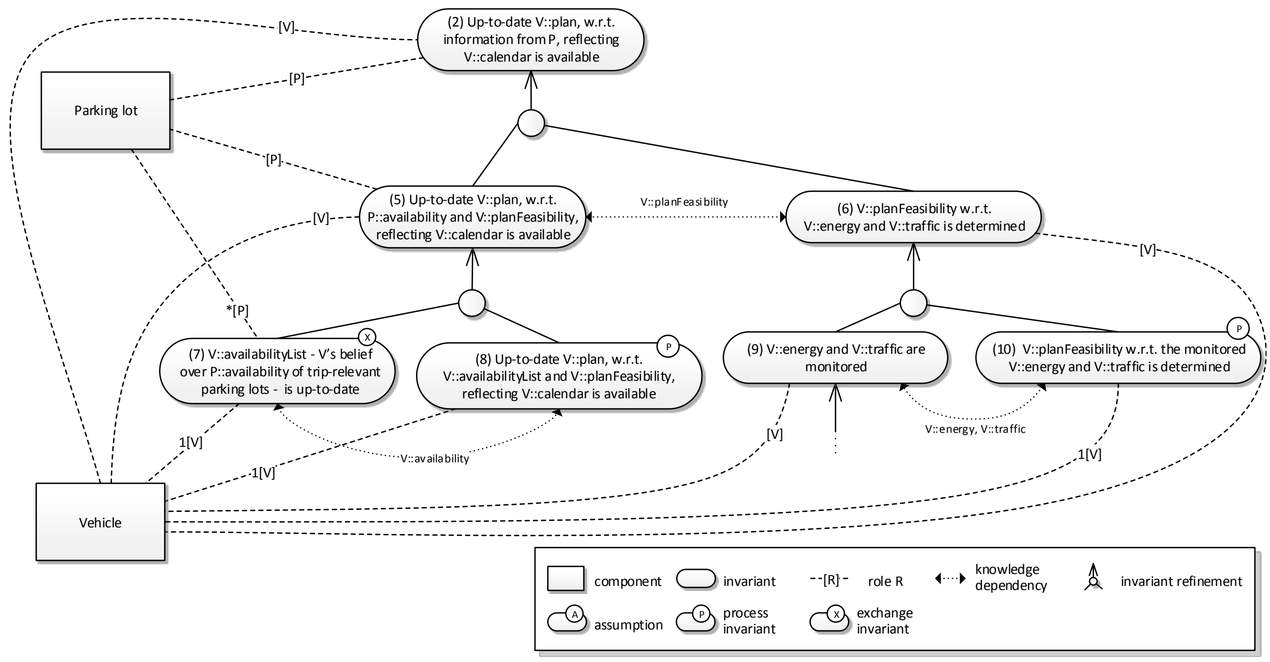
From the dual (computation-activity-based) perspective of refinement, a simultaneous (i.e., parallel) execution of the computation activities associated with the sub-invariants forms the computation activity of the parent. In a refinement with knowledge dependencies, an adequate *scheduling* of these activities is to be determined in the refinement.

4.4 Leaves of Refinement

The rule of thumb is that refinement is finalized when each leaf invariant of the refinement tree is either an assumption or is associated with a "real" computation activity – a *process* or *knowledge exchange*.

Specifically, an invariant that is referring to a single component captures only the operational normalcy to be maintained by a process of the component. Such an invariant is called a *process invariant* (in diagrams marked by P, e.g., (3) in Figure 3).

In a general case when several components take a role in an invariant, e.g., (5) in Figure 5, the situation is more complex. To refine an invariant I_p, referencing the components $C_1, \dots, C_m$ into sub-invariants $I_{s1}, \dots, I_{sn}$ that are eventually associated with "real" computation activities need to apply the concept *belief of C_1 over the knowledge of $C_2, \dots, C_m$*: the belief $B_{C_1}^{C_2,\dots,C_m}(K)$ is knowledge of C_1 that represents C_1's snapshot of a part K of the knowledge of $C_2, \dots, C_m$. For instance, in Figure 5, the belief V::availabilityList of Vehicle over the knowledge P::availability of Parking lots is an example of such a knowledge snapshot (denoted as V::availabilityList = $B_{\text{Vehicle}}^{\text{Parking lot}}$(P::availability)).



(2) Up-to-date V::plan, w.r.t. information from P, reflecting V::calendar is available
[V]
[P]
Parking lot
[P]
(5) Up-to-date V::plan, w.r.t. P::availability and V::planFeasibility, reflecting V::calendar is available
V::planFeasibility
(6) V::planFeasibility w.r.t. V::energy and V::traffic is determined
[V]
[V]
*[P]
X
(7) V::availabilityList - V's belief over P::availability of trip-relevant parking lots - is up-to-date
P
(8) Up-to-date V::plan, w.r.t. V::availabilityList and V::planFeasibility, reflecting V::calendar is available
(9) V::energy and V::traffic are monitored
P
(10) V::planFeasibility w.r.t. the monitored V::energy and V::traffic is determined
1[V]
1[V]
V::availability
[V]
V::energy, V::traffic
1[V]
Vehicle
component
invariant
[R]
role R
knowledge dependency
invariant refinement
A
assumption
P
process invariant
X
exchange invariant

Figure 5: Invariant refinement of "V has an up-to-date V::plan reflecting V::calendar".

Thus, I_{s1} formulates the operational normalcy properties of $B_{C_1}^{C_2,\dots,C_m}$, whereas $I_{s2},\dots,I_{sn}$ refine I_p while substituting the references to the knowledge of $C_2,\dots,C_m$ by references to $B_{C_1}^{C_2,\dots,C_m}$. Note that $B_{C_1}^{C_2,\dots,C_m}$ is a new knowledge introduced into C_1. For example, in Figure 5, (7) formulates the condition on creating the belief V::availabilityList $= B_{\text{Vehicle}}^{\text{Parking lot}}$(P::availability), whereas (8) refines (5) while substituting the references to P::availability by references to V::availabilityList.

As a result, I_{s1} becomes an *exchange invariant* (in diagrams marked by X, such as (7) in Figure 5), since it corresponds to knowledge exchange as its "real" computation activity.

Furthermore, $I_{s2},\dots,I_{sn}$ are potentially process/exchange invariants, since, in general, the number of components taking a role in $I_{s2},\dots,I_{sn}$ is, compared to I_p, decreased at least by one due to references to the belief $B_{C_1}^{C_2,\dots,C_m}$ (such as when comparing (5) and (8) in Figure 5).

4.5 From Invariants to Final Architecture

After the set of components is identified and refinement tree of invariants is completed, the design continues by refining each process invariant into a component process and each exchange invariant into an ensemble. For example, as illustrated in Figure 2, Vehicle is reified by Vehicle0123, while (8) from Figure 5 is refined into its computePlan process and (7) from Figure 5 is refined into the UpdateAvailabilityInformation ensemble. Thus, determined by the invariant refinement, this step yields the final architecture of the system. The details are beyond the scope of this paper; we refer the interested reader to [4].

5. BRIDGING ABSTRACTION LEVELS VIA INVARIANT PATTERNS

While high-level invariants capture general operational normalcy, low-level ones – reflecting architectural elements – capture the EBCS-specific aspects (e.g., periodic scheduling of component processes and knowledge exchange). In this section we elaborate on how to bridge this abstraction gap during refinement. In particular, we describe five patterns of invariants we have identified to reflect the way operational normalcy is captured at four adjacent abstraction levels that bridge this abstraction gap. The contribution lies in the fact that we are able to rigorously describe (and provide guidelines for) the refinement between invariants on the same/adjacent levels of abstraction by assuming that each invariant is an instantiation of a corresponding invariant pattern.

Thus, we can (iteratively) exploit these patterns and guidelines during refinement to continuously lower the level of abstraction until we reach the level of architectural elements. Namely, these patterns are (from the most abstract to the least abstract): (i) *general invariants*, (ii) *present-past invariants*, (iii) *activity invariants*, (iv) *process invariants*, and (v) *exchange invariants* (as an exception, (iv) and (v) are at the same level of abstraction). Figure 6 illustrates the patterns on the case study.

To give a more exact perspective of the patterns, we use a predicate formalization of invariants. Note that in this paper the goal of the formalization is to illustrate the conceptual differences between the patterns rather than to provide their rigorous description, which is beyond the scope of this paper. For formal pattern definition, we refer the interested reader to [6]. Recall that an invariant expresses the operational normalcy in terms of a condition to be maintained during knowledge evolution in time (Section 4.1). Thus, the formalization provides means for referring to timed sequences of knowledge values, which gives a complete view on the knowledge value evolution over time. Specifically, since EBCS-based systems are inherently asynchronous, we are interested in such a formalization that captures the evolution in terms of asynchrony and delays. For example, considering the knowledge evolution illustrated in Figure 7, we are interested in a formalization of the form "*The value of V::pAvailable always equals the value of P::available*

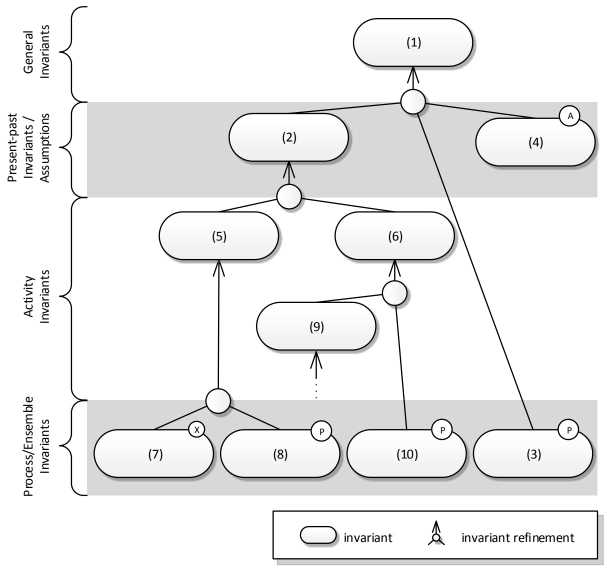


Figure 6: Patterns of invariants in the case study.

that is not older than the period" rather than "*V::pAvailable equals P::available*" (which does not always hold).

Thus, we formalize the invariants as follows. *Time* is represented by a non-negative real number, i.e., $\mathbb{T} \stackrel{\text{def}}{=} \mathbb{R}_0^+$. *Knowledge* is a set $\mathcal{K} = \{k_1, \dots, k_n\}$ of knowledge elements, where the domain of k_i is denoted as V_i. *Knowledge valuation* of an element k_i is a function $\mathbb{T} \to V_i$ which for a time t yields a value of k_i (denoted as $k_i[t]$). An invariant is thus a *predicate* (in a higher-order predicate logic with arithmetic) over a knowledge valuations and time.

Note that in general it is possible to use other forms of formalization; e.g., real-time LTL [2]. However, in this paper the choice of the formalization is driven by the aim of describing invariant refinement rather than model checking. Thus, we consider the proposed predicate formalization more practical (i.e., it is more suitable for formulating and proving relevant theorems).

5.1 General Invariants

General invariants at the top-level of abstraction capture the operational normalcy in terms of relating the past and current knowledge valuation to a future knowledge valuation.

An example of this pattern is the invariant (1) from Figure 3: "All Vehicles meet their calendar", which can be formalized as follows (assuming only a single POI in the calendar, which does not change in time for brevity):

$$\exists t \in \mathbb{T}, t \leq V\text{::calendar.deadline}[0]: \\ V\text{::position}[t] = V\text{::calendar.destination}[0]$$

Note that the invariant does not refer to current time; instead, it refers to a particular time instant in the future.

5.2 Present-past Invariants

Less-general are *present-past invariants* capturing the operational normalcy in terms of the current and/or past knowledge valuations. This reflects the fact (abstracted away at the level of general invariants) that software systems cannot cope with future data, but have to depend on current and/or past data instead. Further, to determine how much of past data is needed, we define the *lag* of a present-past invariant as the maximal distance in the past that is needed to formulate the operational normalcy of the invariant. Similar to real-time software control systems, we assume that the smaller the lag, the bigger precision and robustness; lag equal to 0 denotes an idealized case where the beliefs of all components are always up-to-date and their actions are instant.

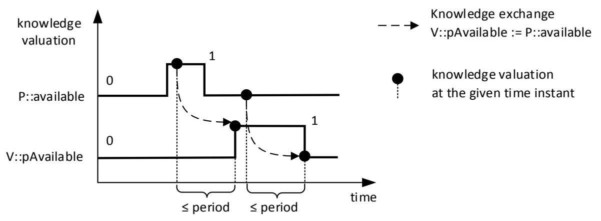


Figure 7: Example of knowledge evolution in time when employing (periodic) knowledge exchange.

An example of this pattern is the invariant (2) from Figure 3: "Up-to-date V::plan, w.r.t. information from P, reflecting V::calendar is available", which can be for parking lots $P_1 \dots P_n$ and a lag L formalized as follows:

"*At any time, for the current valuation of* V::plan *there is a valuation of knowledge of $P_1 \dots P_n$ and* V::calendar *not older than the lag* L *such that they together meet the condition expressed by the* UpToDatePlan *predicate.*"

In the predicate logic, it can be captured as follows:

$$\forall t_{cur} \in \mathbb{T}, \exists t_1, \dots, t_n, t_{cal} \in \mathbb{T}, 0 \leq t_{cur} - t_i \leq L \;\; i \in \{1..n, cal\}: \\ UpToDatePlan(P_1[t_1], \dots, P_n[t_n], V\text{::calendar}[t_{cal}], V\text{::plan}[t])$$

Here, L equal to 0 reflects the case where the V::plan is at each time instant up-to-date with respect to the current knowledge of the parking lots. The bigger L the more outdated parking-lot knowledge valuation is considered.

For all present-past invariants of this syntactic structure, we can use the following shortcut expressing the above-described formalization of (2) from Figure 3 (note, that the "p-p" subscript indicates that this shortcut pertains to the present-past invariant pattern):

$$UpToDatePlan_{p-p}^{L}[P_1, \dots, P_n, V\text{::calendar}][V\text{::plan}]$$

Such a shortcut can be also exploited during invariant refinement for introducing new present-past invariants; it would serve as a "macro" that transforms a time-oblivious predicate (e.g., $UpToDatePlan$) into a formalized present-past invariant of the above-described structure.

5.3 Activity Invariants

Based on the dual concept of computation activities, an *activity invariant* captures the operational normalcy in terms of the current valuation of the output knowledge of the associated computation activity and the current/past valuation of the input knowledge. This follows the idea that a computation activity in EBCS maintains the operational normalcy periodically by reading the input knowledge, performing the computation and writing the output knowledge.

Being relatively low-level, an activity invariant reflects detailed properties of a computation activity that corresponds to software computation. First, it captures the requirement that the output knowledge changes only as a result of the computation activity. Here, we assume that no activities have the same output knowledge. Moreover, an activity invariant captures read consistency of the input knowledge, i.e., that each output knowledge valuation is based on the same or newer input knowledge valuation than the previous one. In an ideal case, the computation is instant, relating thus the current valuation of both the input and output knowledge. Similarly to present-past invariants, the maximal distance in the past needed to formulate the operational normalcy is expressed by the lag of the invariant.

An example of this pattern is the invariant (5) from Figure 5: "Up-to-date V::plan, w.r.t. P::availability and V::planFeasibility, reflecting V::calendar is available", which can be for parking lots $P_1 \dots P_n$ and lag L formalized as follows:

"*There is an execution of the planning activity maintaining the condition* UpToDatePlan *such that at any time the valuation of* V::plan *corresponds to the outcome of the activity applied on the valuation of the input knowledge* P::availability, V::planFeasibility, *and* V::calendar *not older than lag* L. *Moreover, each valuation of* V::plan *is based on newer valuation of the input knowledge than the previous one.*"

In the predicate logic, it can be captured as follows:

$$\exists a_1, \dots, a_n, a_{pF}, a_{cal} \colon \mathbb{T} \to \mathbb{T},$$
$$0 < x - a_i(x) \le L \; \forall i \in \{1..n, pF, cal\},$$
$$a_i(x) \le a_i(y) \; \forall x, y \colon x \le y \; \forall i \in \{1..n, pF, cal\},$$
$$\forall t \in \mathbb{T} \colon$$
$$UpToDatePlan\begin{pmatrix} P_1\text{::availability}[a_n(t)], \\ \vdots \\ P_n\text{::availability}[a_n(t)], \\ V\text{::planFeasibility}[a_{pF}(t)], \\ V\text{::calendar}[a_{cal}(t)] \\ V\text{::plan}[t] \end{pmatrix}$$

Here, the usage of a non-decreasing function $a_i \colon \mathbb{T} \to \mathbb{T}$ rather than a particular $t_i \in \mathbb{T}$ captures the read consistency and the fact that V::plan may change only as the result of an execution of a planning activity.

Again, L equal to 0 reflects the case where the valuation of V::plan is at each time instant up-to-date with respect to the current valuation of P::availability of the parking lots and V::planFeasibility of the vehicle. In other words, the associated computation activity computes infinitely fast and infinitely often. The bigger L the more outdated valuation of P::availability and V::planFeasibility is considered; i.e., the slower/less often is the computation activity expected to execute.

Similar to present-past invariants, the shortcut for the above-described formalization of (5) from Figure 5 is:

$$UpToDatePlan_{act}^{L}\begin{bmatrix} P_1\text{::availability}, \\ \vdots \\ P_n\text{::availability}, \\ V\text{::planFeasibility}, \\ V\text{::calendar} \end{bmatrix}\begin{bmatrix} V\text{::plan} \end{bmatrix}$$

5.4 Process invariants

Refining an activity invariant at the lowest level of abstraction, an invariant may take the form of a process invariant – referring to a single component, capturing the operational normalcy to be maintained by a (periodic) process of the component (Section 4.4).

Such an invariant captures detailed properties of the periodic scheduling of the process. The difference to activity invariants lies in the fact that not only the output knowledge valuation may change as a result of performing the computation activity alone and must be based on current-enough input knowledge valuation, but also that the computation activity is performed exactly once in each period. In this context, the period is an elaboration of the activity-predicate lag. Specifically, since we assume a component process to be periodic and (soft) real-time, the output knowledge valuation is determined by the release time and finish time of the process in each period [7].

An example of this pattern is the invariant (8) from Figure 5: "Up-to-date V::plan, w.r.t. V::availabilityList and V::planFeasibility, reflecting V::calendar is available", which can be for period L formalized as follows:

"*If the current time is before the finish time of the process in the current period, then the* V::plan *valuation is the same as in the previous period; i.e., it corresponds to the outcome of the process w.r.t. the inputs* V::availabilityList, V::planFeasibility, *and* V::calendar *at the release time of the process in the previous period. Otherwise,* V::plan *corresponds to the outcome of the process w.r.t. the inputs at the release time in this period.*"

In the predicate logic, it can be captured as follows:

$$\exists R, F \colon \mathbb{N} \to \mathbb{T}, P(x-1) \le R(x) < F(x) < P(x),$$
$$\forall p \in \mathbb{N}, \forall t \in \langle P(p-1), P(p)) \colon$$
$$t < F(p) \Rightarrow UpToDatePlan\begin{pmatrix} V\text{::availabilityList}[R(p-1)], \\ V\text{::planFeasibility}[R(p-1)], \\ V\text{::calendar}[R(p-1)], \\ V\text{::plan}[t] \end{pmatrix}$$
$$t \ge F(p) \Rightarrow UpToDatePlan\begin{pmatrix} V\text{::availabilityList}[R(p)], \\ V\text{::planFeasibility}[R(p)], \\ V\text{::calendar}[R(p)], \\ V\text{::plan}[t] \end{pmatrix}$$

where $P(n) \colon \mathbb{N}_0 \to \mathbb{T} = n * L$; i.e., the end of the n-th period. $R(n)$ and $F(n)$ denote the release and finish time of the real-time process in the n-th period.

Here, L approaching 0 reflects the case, where the V::plan is at each time instant infinitely close to the up-to-date plan with respect to the current V::availabilityList, V::planFeasibility, and V::calendar of the vehicle.

Again, the shortcut for the above-described formalization of (8) from Figure 5 is:

$$UpToDatePlan_{proc}^{L}\begin{bmatrix} V\text{::availabilityList}, \\ V\text{::planFeasibility}, \\ V\text{::calendar} \end{bmatrix}\begin{bmatrix} V\text{::plan} \end{bmatrix}$$

5.5 Ensemble invariants

An activity invariant may at the lowest level of abstraction be refined also into an ensemble invariant – capturing the operational normalcy to be maintained by (periodic) knowledge exchange of an ensemble among the referred components (Section 4.4).

Such an invariant captures detailed properties of the periodic scheduling of knowledge exchange. Compared to process invariants, an exchange invariant further accounts for the delay connected with potential transfer of the knowledge over the network (as required in distributed systems). The invariant thus describes a composite computation activity consisting of the knowledge transfer (with an upper time bound on its duration) followed by periodic evaluation of the membership condition and the knowledge exchange. Further, it is assumed that such composite activities may be partially overlapping (mostly in situations when the knowledge transfer takes longer than the period of the knowledge exchange).

An example of this pattern is the invariant (7) from Figure 5: "V::availabilityList – V's belief over P::availability of trip-relevant parking lots – is up-to-date", which can be for parking lots $P_1 \dots P_n$, period L, and upper bound for knowledge transfer T formalized as follows:

"*If the current time is before the finish time of the knowledge exchange for* V *in the current period, then the* V::availabilityList *valuation is the same as in the previous period. Otherwise,* V::availabilityList *equals the set of* P::availability *for all relevant P_i as available at* V *at the release time in this period. It takes at most T for the knowledge of P_i to become available at* V*. Further always the newest knowledge of P_i is taken into account.*"

In the predicate logic, it can be captured as follows:

$$\begin{gathered}\exists a_1, \dots, a_n, : \mathbb{T} \to \mathbb{T},\\ 0 < x - a_i(x) \le T\ \forall i \in \{1..n\},\\ a_i(x) \le a_i(y)\ \forall x, y : x \le y\ \forall i \in \{1..n\},\\ \exists R, F : \mathbb{N} \to \mathbb{T}, P(x-1) \le R(x) < F(x) < P(x),\\ \forall p \in \mathbb{N}, \forall t \in \langle P(p-1), P(p)\rangle :\end{gathered}$$

$$t < F_V(p) \Rightarrow EqualsRelevant\begin{pmatrix} P_1\text{::availability}[a_1(R(p-1))],\\ \vdots\\ P_n\text{::availability}[a_n(R(p-1))],\\ V\text{::availabilityList}[t]\end{pmatrix}$$

$$t \ge F_V(p) \Rightarrow EqualsRelevant\begin{pmatrix} P_1\text{::availability}[a_1(R(p-1))],\\ \vdots\\ P_n\text{::availability}[a_n(R(p-1))],\\ V\text{::availabilityList}[t]\end{pmatrix}$$

where $P(n): \mathbb{N}_0 \to \mathbb{T} = n * L$; i.e., the end of the n-th period. $R(n)$ and $F(n)$ denote the release and finish time of the real-time knowledge exchange in the n-th period. Finally, $a_i(t)$ denotes the time at which the value of knowledge from P_i that is available at V at time t has been sent to V.

Here, L approaching 0 reflects the case, where the V::availabilityList is at each time instant infinitely close to the set of the current P::availability of all the relevant parking lots.

The shortcut for the above-described formalization of (7) from Figure 5 is:

$$EqualsRelevant_{ens}^{L,T}\begin{bmatrix} P_1\text{::availability,}\\ \vdots\\ P_n\text{::availability}\end{bmatrix}\begin{bmatrix} V\text{::availabilityList}\end{bmatrix}$$

5.6 Refinement among Invariant Patterns

Having described the invariant patterns, we will now briefly elaborate on the refinement between invariants following the patterns on the same/adjacent levels of abstraction in order to provide guidelines for decomposition. In particular, we list the expected variants of decomposition and discuss when each of the variants is a refinement. This can serve as guidelines during decomposition at the corresponding levels of abstraction in order to guarantee refinement. Note that the claims below are articulated in an informal way, while formal proofs can be found in [6].

General→Present-past. At the top level of abstraction, during refinement of a general invariant into a conjunction of present-past invariants, it is necessary to introduce assumption invariants (e.g., (4) in Figure 3). Technically, these assumptions are necessary to guarantee that maintaining the operational normalcy based on the current and/or past knowledge valuation will eventually result in reaching the operational normalcy based on a future knowledge valuation. The correctness of this step has to be proved for each case separately (e.g., via a theorem prover), which makes it the most demanding from the formal point of view.

Present-past→Present-past. In a refinement of one present-past invariant by means of other present-past invariants, it holds that the combined lag of the sub-invariants is lesser or equal to the parent's lag. The combination is determined by the knowledge dependencies among the sub-invariants.

Present-past→Activity. It holds that the activity invariant pattern is a strict refinement of the present-past invariant pattern; i.e., $P_{act}^{L}[I][O] \Rightarrow P_{p-p}^{L}[I][O]$ for each P, I, and O.

Activity→Activity. The refinement of one activity invariant by means of other activity invariants is similar to the case present-past→present-past. For our predicate formalization, it is possible to determine this form of refinement solely based on the time-oblivious skeletons of the invariants and the structure of the decomposition (i.e., without interpreting the full invariants via a theorem prover).

Activity→Process. It holds that the process invariant pattern is a refinement of the activity invariant pattern with lag equal twice the period of the process invariant pattern; i.e., $P_{proc}^{L}[I][O] \Rightarrow P_{act}^{2L}[I][O]$ for each P, I, and O. This complies with the well-known fact in the area of real-time scheduling: in order to achieve a particular end-to-end response time with a real-time periodic process with relative deadline equal to period, the period needs to be at most half of the response time [7].

Activity→Exchange. Similarly, it holds that the exchange invariant pattern is a refinement of the activity invariant pattern with lag equal twice the period of the exchange invariant pattern plus the time for distributed transfer of the knowledge; i.e., $P_{ens}^{L,T}[I][O] \Rightarrow P_{act}^{2L+T}[I][O]$ for each P, I, and O.

6. EVALUATION AND DISCUSSION

6.1 Case Study

To evaluate IRM, we have employed it during design of the case study. As a final step, we have successfully validated the resulting EBCS/DEECo architecture by implementing it in the jDEECo component framework[1]. Since the detailed models created within the study are proprietary, we present only a summary and lessons learned. For a concise version of the case study, which includes detailed design, we refer the reader to [23].

While having a single top-level goal, the design included 2 components and 20 invariants in total. In particular, 4 of them were exchange invariants, 8 process invariants, 2 present-past invariants, and the other 5 (excluding the top-level goal) activity invariants.

Eventually, the design led to an EBCS/DEECo architecture consisting of 4 ensembles among the 2 components, where one component constituted 3 processes maintaining 6 process invariants, while the other component constituted 1 process maintaining 2 process invariants.

As a significant benefit, not only we were able to gradually design a desired architecture (which could be in fact potentially obtained using conventional design methods), but the invariant decomposition tree also constituted a "proof of correctness" of the design with respect to the top-level goal.

Although IRM is in general a top-down process, the important lesson learned from the case study was that refinement is inherently too complex to be done correctly just this way. Thus, several iterations, series of top-down and bottom-up steps, had to be performed to get a satisfactory design.

[1] The current implementation of jDEECo is available at https://github.com/d3scomp/JDEECo

6.2 Correctness by Construction

So far, we have used the predicate formalization only to illustrate the individual invariant patterns. However, if applied consistently throughout the whole design, it would be possible to formally verify each of the refinement steps in support of achieving correctness by construction.

An obvious obstacle of verification of such a complete predicate formalization is that the predicate logic we use is fairly complex (continuous time, quantifiers over function symbols, etc.). Thus, verification via a theorem prover is not a viable option due to lack of efficiency.

Nevertheless, as already indicated in Section 5.6, correctness of particular kinds of refinement can be decided without interpreting full invariants via a theorem prover. To date, we have formulated and proved a theorem deciding correctness of activity→activity predicate refinement. In particular, we have been focusing on so called "flow decomposition" [6] where the sub-invariants constitute a simple pipe-and-filter architecture (i.e., the kind of decomposition used in the examples of Sections 4 and 5).

6.3 Runtime Verification

Unfortunately, not all forms of refinement can be verified via application of theorems (e.g., general→present-past refinement). The correctness of such refinement can, however, be addressed by runtime verification. Although this does not provide design-time assurances, it at least helps in detection and localization of design errors.

An important feature of IRM with respect to runtime verification is that IRM refinement hierarchy actually over-specifies the system-to-be. This is because there is an *implies* relationship between the sub-invariants and the parent invariant in a refinement (recursively up to the top-level invariant). However, at runtime it is possible to evaluate not only the lower-level invariants but also the parent. This allows distinguishing different types of errors from unexpected behavior. In particular, given an invariant I and its refinement into $I_1, \ldots, I_n$ (which means that by definition $I_1, \ldots I_n \Rightarrow I$), we can distinguish 4 different cases:

(1) All $I_1, \ldots, I_n$ hold and I holds: Correct operation of the system.
(2) All $I_1, \ldots, I_n$ hold and I does not hold: Error in design – mostly because of neglecting a hidden assumption in refinement of I.
(3) At least one $I_1, \ldots, I_n$ does not hold and I holds: Potential for improvement of the design – refinement of I is likely to have more strict assumptions than necessary.
(4) At least one $I_1, \ldots, I_n$ does not hold and I does not hold: Incompatible environment – this particular refinement of I cannot be used in the current environment.

Obviously a modification of the design may be needed when any of cases (2) – (4) has been detected. However, the goals of the redesign are different. While in (2) it is for correcting an obvious error, in (3) it is to generalize the design and in (4) it is to either extend the design or provide another design alternative suitable for a given environment.

6.4 Novelty and Benefits

The strength of IRM lies in the fact that it directs reasoning along the lines of what needs to hold at every time instant (expressed via invariants) as opposed to what needs to be performed (actions) or what should hold in the future (goals). Thus, it allows expressing the relation of a component to its environment and itself, which is particularly valuable for the design of autonomous adaptive RDS that continuously interact with their environment to achieve the desired goals.

Technically, IRM is novel in employing ensembles as a systematic foundation for capturing knowledge interdependence (logical and temporal) of otherwise autonomous components. This allows keeping an appropriate level of abstraction and separation of concerns when designing a component for an adaptive and autonomous operation. In particular, IRM benefits from recursive step-by-step top-down decomposition with precise refinement semantics. The refinement semantics is special in the sense that it reflects operational and communication delays (inherent to actual RDS implementations) by exploiting the concepts of belief and knowledge exchange.

7. RELATED WORK

The iterative refinement of invariants found in IRM is reminiscent of goal-oriented requirements analysis from the field of requirements engineering [22]. In particular, the Keep All Object Satisfied (KAOS) method [20] is a well-established method for capturing and analyzing system requirements in form of goals, assumptions, and domain properties. The idea is to decompose the abstract high-level goals into more concrete sub-goals up to the level where goals represent requirements that can be handled by individual system agents. Since goals can be formulated in first-order linear temporal logic [2], the goal model can be formally checked for consistency and completeness [20]. Pre-defined, verified patterns can also be used to guide the goal decomposition process [11]. A similar approach is employed within Tropos method [8], where goals, soft-goals, tasks and dependencies are modeled and analyzed from the perspective of the autonomous agents. However, these models either do not map effectively to the later development phases (KAOS), or do not support mapping to emergent architectures (Tropos), which are typical in EBCS [13].

Recent work in requirements modeling specifically targeting the domain of EBCS has been carried out within the scope of the ASCENS project and has been integrated into the Statement of the Affairs (SOTA) [1] and POEM [15] models. The key idea of SOTA is to abstract the behavior of a system with a single trajectory through a state space, which represents the set of all possible states of the system at a single point of time. The requirements of a system in SOTA are captured in terms of goals. A goal is an area of the SOTA space that a system should eventually reach, and it can be characterized by its pre-condition, post-condition, and utilities. Thus SOTA provides the means to capture the early requirements of different component cooperation schemes. IRM, on the other hand, stands as an intermediate method, which guides the transition from early (high-level) requirements to system architecture in terms of components and ensembles.

8. CONCLUSION AND FUTURE WORK

In this paper, we have presented a novel Invariant Refinement Method (IRM), targeting architectural design of Resilient Distributed Systems (RDS) by building on the concepts of Ensemble-Based Component Systems (EBCS). IRM is a systematic design method which starts with the overall system goal and ends up by establishing a system architecture composed of components and ensembles. Building on goal-based requirements elaboration, IRM integrates additional aspects such as architecture refinement and (soft) real-time scheduling.

IRM raises a number of interesting questions for further research. In particular, they include: (i) providing a formal framework (i.e., definitions and theorems) for deciding correctness of refinement within a suitable predicate formalization of invariants, (ii) focusing on RDS with respect to changes in the environment on efficient representation of the environment during the design; (iii) thoroughly exploring the application of IRM for runtime verification. Also, as a future work, we aim at obtaining automated tools for IRM that would help guide design decisions during refinement and check correctness of the resulting design. These include technical tools for checking (syntactic) consistency of the design, as well as tools exploiting a formal framework and/or employing formal reasoning for checking (semantic) correctness.

9. ACKNOWLEDGMENTS

This work was partially supported by the EU project ASCENS 257414 and the Grant Agency of the Czech Republic project P103/11/1489. The work was also partially supported by Charles University institutional funding SVV-2013-267312.

10. REFERENCES

[1] D.B. Abeywickrama, N. Bicocchi, and F. Zambonelli. SOTA: Towards a General Model for Self-Adaptive Systems. In *Proc. of WETICE '12*, 2012.

[2] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In *Proc. of FSTTCS '06*, 2006.

[3] F. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley, 2007.

[4] T. Bures, I. Gerostathopoulos, V. Horky, J. Keznikl, J. Kofron, M. Loreti, and F. Plasil. *Language Extensions for Implementation-Level Conformance Checking*. ASCENS Deliverable 1.5. Available at: http://www.ascens-ist.eu/deliverables, 2012.

[5] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil. *DEECo – an Ensemble-Based Component System*. In *Proc. of CBSE 2013*, ACM, 2013.

[6] T. Bures, I. Gerostathopoulos, J. Keznikl, and F. Plasil. *Formalization of Invariant Patterns for the Invariant Refinement Method*. Technical Report no. D3S-TR-2013-04. D3S, Charles University in Prague. Available at: http://d3s.mff.cuni.cz/publications, 2013.

[7] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency*, ser. Series in Computer Science, R. G. Melhem, Ed. Springer US, 2005.

[8] J. Castro, M. Kolp, L. Liu, and A. Perini. Dealing with Complexity Using Conceptual Models Based on Tropos. In *Conceptual Modeling: Foundations and Applications*. Ser. LNCS, Springer Berlin, Heidelberg, vol. 5600, 2009.

[9] I. Crnkovic. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[10] I. Crnkovic, M. Chaudron, and S. Larsson. Component-based development process and component lifecycle. *Software Engineering Advances*, 44, 2006.

[11] R. Darimont, and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Proc. of SIGSOFT '96*, 1996.

[12] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese. A Language-based Approach to Autonomic Computing. In *Proc. of FMCO '11*, 2012.

[13] I. Gerostathopoulos, T. Bures, and P. Hnetynka. Position Paper: Towards a requirements-driven design of ensemble-based component systems. In *Proc. of International Workshop on Hot Topics in Cloud Services, ICPE '13*, 2013.

[14] M. Holzl, A. Rauschmayer, and M. Wirsing. Engineering of software-intensive systems: State of the art and research challenges. In *Software-Intensive Systems and New Computing Paradigms*. Ser. LNCS, Springer Berlin, Heidelberg, vol. 5380, 2008.

[15] M. Holzl, et al. Engineering Ensembles: A White Paper of the ASCENS Project. *ASCENS Deliverable JD1.1.* Available at: http://www.ascens-ist.eu/whitepapers, 2011.

[16] M. C. Huebscher and J. A. McCann. A survey of autonomic computing–degrees, models, and applications. *ACM Computing Surveys*, 40, 3, 2008.

[17] IBM. An architectural blueprint for autonomic computing. *IBM White Paper*, 2003.

[18] N. R. Jennings. On agent-based software engineering. *Artificial intelligence*. 117, 2000.

[19] J. Keznikl, T. Bures, F. Plasil, and M. Kit. Towards Dependable Emergent Ensembles of Components: The DEECo Component Model. In *Proc. of WICSA/ECSA 2012*, IEEE CS, 2012.

[20] A. Lamsweerde. Requirements engineering: from craft to discipline. In *Proc. of SIGSOFT '08/FSE-16*, 2008.

[21] A. Rao, and M.P. Georgeff. BDI agents: From theory to practice. In *Proc. of ICMAS '95*, 1995.

[22] N. U. Rehman, S. Bibi, S. Asghar, and S. Fong. Comparative Study of Goal-Oriented Requirements Engineering. In *Proc. of NISS '10*, 2010.

[23] N. Serbedzija, et al. *Ensemble Model Syntheses with Robot, Cloud Computing and e-Mobility.* ASCENS Deliverable 7.2. Available at: http://www.ascens-ist.eu/deliverables, 2012.

[24] Y. Shoham, and K. Leyton-Brown. *Multiagent Systems: Algorithmic, GameTheoretic, and Logical Foundations,* Cambridge University Press, 2008.

[25] J. A. Stankovic , T. He , T. Abdelzaher , M. Marley , G. Tao, S. Son , and C. Lu. Feedback control scheduling in distributed real-time systems. In *Proc. of RTSS '01*, 2002.

[26] E. Vassev, and M. Hinchey. The Challenge of Developing Autonomic Systems. *Computer*, 43, 12, 2010.

3.4 Automated Resolution of Connector Architectures Using Constraint Solving (ARCAS method)

Jaroslav Keznikl,
Tomáš Bureš,
František Plášil,
Petr Hnětynka

In the **Software & Systems Modeling** journal.

Volume 13, Issue 2,
published by Springer Berlin Heidelberg,
pages 843-872,
print ISSN 1619-1366,
online ISSN 1619-1374,
May 2014.

The original version is available electronically from the publisher's site at http://dx.doi.org/10.1007/s10270-012-0274-8.

Summary of the Paper

This paper, published as [KBPH14], addresses the challenge of automated synthesis of software connectors that would enable building dependable, scalable, and open-ended distributed software architectures with heterogeneous deployments (i.e., tackling the challenges C3 and C4 outlined in Section 1.3). Moreover, the paper opts for supporting synthesis at runtime, thus providing a potential technical baseline for tackling the challenge of recurrent architecture adaptation (i.e., C1).

Attacking the research goal **G2**, this paper responds to these challenges by presenting ARCAS (Automated Resolution of Connector Architectures using constraint Solving) – a novel, formally grounded method for open-ended design of middleware-based connectors and their automated synthesis at deployment time/runtime. It allows reflecting both (i) the connector-specific concerns [BP04, IBB11, TMD10], including the communication style and non-functional properties (NFPs), and (ii) the current connector context, including particular properties of the connected components and the current deployment.

In principle, ARCAS is based on an automated composition of connectors from reusable *elements* that capture the common connector-implementation patterns related to a particular communication style, middleware, and NFPs. Specifically, given a design specification of a connector, including NFPs and deployment requirements imposed on the components being connected, ARCAS produces a connector instance configuration (CIC) that describes a particular composition (and parameterization) of elements to realize the connector. From a big-picture perspective, ARCAS is the first phase in a two-step connector generation process [Bur06]. In the second step, CIC is used as the input for the actual code generation process [MPBH13] yielding a deployable connector code.

The key idea of ARCAS is to employ constraint solving for automated construction of CIC. In particular, ARCAS employs the Alloy modeling language [Jac02, Jac12][1] to capture the predefined elements, as well as the design and deployment requirements of a connector, into a formal logic theory – connector theory. Then, it employs the model-finding feature of Alloy Analyzer to find a model of the connector theory (in the sense of logic), representing a CIC for the specified connector. Semantically, a model of a connector theory represents both a correct CIC (the elements in the CIC are able to cooperate according to their predefined specification), and a desired CIC (the corresponding connector complies with the design and deployment requirements). Note that ARCAS is in the paper actually described in terms of first-order finite structures (i.e., first-order logic predicates upon finite sets and relations) and Alloy is used only as a concrete representative that provides a convenient syntax and tooling.

To avoid manual construction of a connector theory, ARCAS uses a model-based approach for obtaining the connector theory in an automated way by transformation of a connector specification.

Building on previous work [BP04, Bur06], ARCAS facilitates separation of concerns and enables open-ended connector design by separating the architecture-design perspective and connector-design perspective of a connector specification. The former focuses on the design and properties of a connector in the context of a particular component-based architecture, relying on the concepts generally agreed in the area of middleware-based connectors [CL02] (e.g., communication style, required NFPs, association of connector endpoints with component interfaces, concrete deployment, the supported middleware, etc.). The latter focuses strictly on the internal connector design and implementation via hierarchical composition of reusable, parametric elements, anticipated for reuse, and is rather specific to ARCAS (e.g., realization of communication styles, internal architecture of the elements, delegation of NFP support across element hierarchy, etc.). The main benefit of this separation is that the architecture-design perspective is open-ended and can be supplemented in an automated way with a proper connector implementation based on the required deployment, NFPs, etc.

In this context, one of the specific contributions of this paper is that it thoroughly elaborates both perspectives. It also provides a concrete and abstract syntax for both perspectives (facilitating the automated transformation into a connector theory), as well as a precise formal semantics given by the transformation into first-order finite structures and Alloy.

Going into more detail, ARCAS describes a CIC in a connector theory using a meta-model expressed in terms of sets and relations. Consequently, both the architecture-design perspective and connector-design perspective of a connector specification are represented as constraints on this meta-model. Given such constraints, while some of the meta-model sets and relations forming the CIC are fully determined (e.g., the ones related to the architecture-design perspective), some are underspecified and, therefore, determined only partially (e.g., the ones related to vertical composition of connector elements and valuation of the elements' parameters). The problem of finding a CIC is then interpreted as finding a realization of the underspecified sets and relations. If multiple alternatives exist, an optimal one is chosen (the paper discusses several ways of achieving this).

The main body of the paper is focusing on a rigorous description of (i) the concrete and abstract syntax for a connector specification, (ii) the transformations of the specification into a connector theory in terms of parameterized templates (relying on first-order finite structures), and (iii) the representation of a connector theory in Alloy.

Comments on Authorship

Although the key idea of the paper builds on previous work and the paper is of equal authorship, I have contributed with a thorough elaboration of the connector specification and its concrete and abstract syntax. Besides, I also elaborated in detail the initial idea of representing a connector theory in Alloy. Further, I personally contributed with the intermediate encoding into first-order finite structures. Moreover, I have conducted the evaluation. Finally, under the indispensable guidance and in collaboration with the other authors, I authored a majority of the text.

Softw Syst Model (2014) 13:843–872
DOI 10.1007/s10270-012-0274-8

REGULAR PAPER

Automated resolution of connector architectures using constraint solving (ARCAS method)

Jaroslav Keznikl · Tomáš Bureš · František Plášil · Petr Hnětynka

Received: 11 August 2011 / Revised: 28 June 2012 / Accepted: 23 July 2012 / Published online: 9 September 2012

Abstract In current software systems, connectors play an important role by encapsulating the communication and coordination logic. Since they share common patterns (elements) depending on characteristics of the connections, the elements can be predefined and reused. A method of connector implementation based on a composition of predefined elements naturally comprises two steps: resolution of the connector architecture, and creation of the actual connector code based on the architecture. However, manual resolution of a connector architecture is very difficult due to the number of factors to be considered. Thus, the challenge is to come up with an automated method, able to address all the important factors. In this paper, we present a method for automated resolution of connector architectures based on constraint solving techniques. We exploit a propositional logic with relational calculus for defining a connector theory, a constraint specification reflecting both the predefined parts and the important resolution factors, and employ a constraint solver to find a suitable connector architecture as a model of the theory. As a proof of the concept, we show how the theory can be captured in the Alloy language and resolved via the Alloy Analyzer.

Communicated by Prof. Marsha Chechik.

J. Keznikl · T. Bureš · F. Plášil (✉) · P. Hnětynka
Faculty of Mathematics and Physics, Charles University in Prague, Malostranske Namesti 25, 118 00 Prague 1, Czech Republic
e-mail: plasil@d3s.mff.cuni.cz

J. Keznikl
e-mail: keznikl@d3s.mff.cuni.cz

T. Bureš
e-mail: bures@d3s.mff.cuni.cz

P. Hnětynka
e-mail: hnetynka@d3s.mff.cuni.cz

Keywords Software architecture · Software connectors · Constraint solving · Middleware-based connectors · Connector theory · Alloy

1 Introduction

Proposed with the aim of supporting the separation of concerns, software connectors are entities solely encapsulating communication and coordination among components, as described, e.g., in [29,38]. In particular, connectors ensure distribution of communicating components [7] while encapsulating middleware (*middleware-based* connectors), provide adaptation in order to achieve middleware-level [19,32] and application-level [10,21,36] interoperability (*adaptors*), and ensure synchronization of component communication [9,21] (*coordinators*). In this paper, we focus particularly on the middleware-based connectors.

Although the introduction of middleware-based connectors provides benefits in terms of separation of concerns and abstraction of particular middleware, it does not necessarily simplify the code development effort, since, in principle, the middleware-related code is moved from components to connectors. In the component models that include connectors, e.g., [34,38], the connector lifecycle differs from the component lifecycle: although partially specified connectors are employed during the application design phase, fully specified connectors emerge at the earliest in the component deployment phase—after decisions on application architecture and deployment have been made. Moreover, deployment of a particular component application may vary from time to time, so that several variants of a connector may be required. Advantageously, these variants typically share common patterns related to a particular communication style [7,19,38], middleware, and non-functional properties (NFPs); therefore,

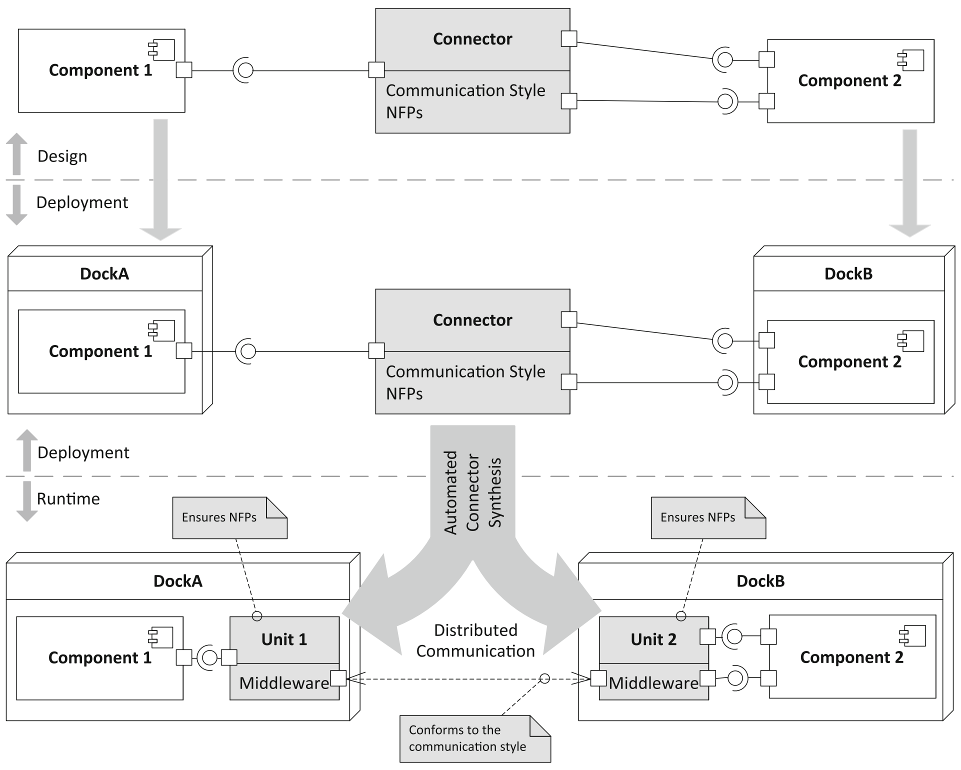


Fig. 1 Role of automated connector synthesis in connector lifecycle

related parts of connectors can be predefined/designed in advance.

Middleware-based connectors can also emerge later, even after some of the components are already running. This is particularly true in the case of independently deployed components available as services (e.g., web services). In principle, the task of a newly emerging connector is to mediate a client component's communication with such a service while respecting the particular middleware employed by the service. In this sense, services are middleware-aware components. Similarly, the emergence of such connectors can be desirable at runtime once architecture and deployment reconfiguration takes place (e.g., due to load balancing).

In this context, the challenge is to find an automated method for synthesis of middleware-based connectors at deployment time/run time, i.e., synthesis of *emergent connectors* [20], in such a way, that reuse of predefined connector parts is maximized. A related issue is to structure the predefined parts accordingly. Another challenge is to support NFPs in the actual connector synthesis and to structure the predefined parts in order to efficiently and flexibly capture variability of NFP requirements. The context of such automated connector synthesis in the design, deployment, and runtime phases of an application is illustrated in Fig. 1.

1.1 Goal of the paper

The goal of this paper is to respond to the challenges above by introducing the ARCAS method (Automated Resolution of Connector Architectures using a constraint-Solving technique). In general, ARCAS is based on an automated composition of connectors from predefined hierarchical *elements* [7,34]. It produces a description of a hierarchical composition of elements reflecting the connector design and deployment requirements imposed on the components being connected.

Technically, given a design specification of a connector, including NFPs and decision on component deployment, ARCAS produces a connector instance configuration (CIC), describing a particular composition of available elements to realize the connector. From a big-picture perspective, ARCAS is the first phase in a two-step connector generation process [2]. In the second step, CIC is used as the input for the actual code generation process [30] yielding a deployable connector code.

The basic idea of ARCAS (Fig. 2) is to employ a constraint-solving technique for automated resolution of CIC. For this purpose, we employ the Alloy modeling language [22–24] for capturing a *connector theory*, i.e., a logic theory playing the role of a constraint specification reflecting both the predefined elements, and the design and deployment

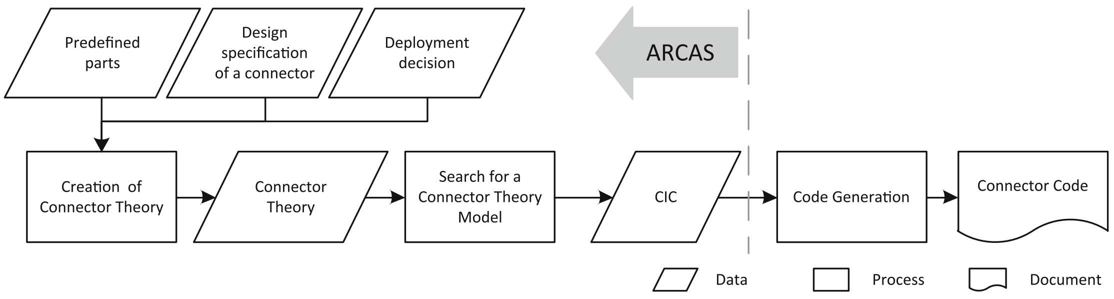


Fig. 2 ARCAS overview

requirements of the connector. Further, we employ the Alloy Analyzer as a constraint solver to find a model of the theory, representing a desired CIC. As an aside, the ARCAS method has applicability also in other domains as discussed further (Sect. 8).

1.2 Structure of the paper

The paper is structured as follows: Sect. 2 presents the basic concepts of middleware connectors and illustrates these with an example. Section 3 gives a brief overview of the whole ARCAS method. Section 4 describes both abstract and concrete syntax of a middleware-connector specification. Section 5 describes the construction of a connector theory in terms of predicate logic and relational calculus. Section 6 provides a brief introduction to the Alloy modeling language and describes ARCAS in terms of Alloy. Section 7 surveys the related work, whereas Sect. 8 provides evaluation and discussion of the ARCAS method, and Sect. 9 concludes the paper while suggesting future work activities.

2 Middleware connectors: basic concepts

In this section, we introduce the basic concepts of middleware connectors. A connector can be viewed from: (i) application designer perspective (*requirements view* and *deployment view*) and (ii) connector designer perspective (*design view*). While (i) focuses on the high-level task and properties of a connector in a particular component application, (ii) focuses strictly on the connector design and implementation. Here, the concepts in (i) are generally agreed in the area of middleware-based connectors [11], whereas the concepts in (ii) stem from our experience with connector design and implementation [7] and are thus rather specific to ARCAS. We recall and illustrate all the concepts on a simple example—a fragment of a distributed component-based application [18] featuring the components `CashDesk` and `Inventory` bound together by a single connector. This setting will be used as a running example throughout the text.

2.1 Application designer perspective

2.1.1 Requirements view

From the application designer perspective, the requirements view of a connector (Fig. 3a) focuses on describing the required NFPs and the communication style of a component connection. As to NFPs (e.g., logging of the connector invocations, secure distributed communication), only those which can be represented via structured/enumerable values are considered. The representation of a NFP is called *feature* (e.g., the `Logging` feature in Fig. 3a has the structured `ToFile` value comprising the `file-name` parameter). As for the communication style, it defines *roles*—the connector's endpoints for communicating with components [11], e.g., the procedure-call communication style defines the roles `Caller` and `Callee`. The components to be connected communicate using instances of the roles. Therefore, in the requirements view of a connector each role instance has to reflect the associated component interface, e.g., the instance of the role `Caller` is associated with the `CashDesk's` required interface, and the instance of the role `Callee` is associated with the Inventory's provided interface.

2.1.2 Deployment view

At the deployment time, a connector has to reflect the actual deployment decision on the connected components. This is the focus of the deployment view of a connector (Fig. 3b). In the example, the distributed environment consists of two component containers (*deployment docks*) A and B. Thus, a connector is viewed as an assembly of distributed connector *units*. The key purpose of a unit is to refine the role instances associated with a particular component in such a way that the communication between the unit and the corresponding component is local, whereas the communication among units is (typically) remote. For instance, the connector defined in Fig. 3b is split into two units, attached to `CashDesk` and `Inventory` components, respectively. In compliance with the desired component deployment, units have to conform to the *capabilities* of the selected

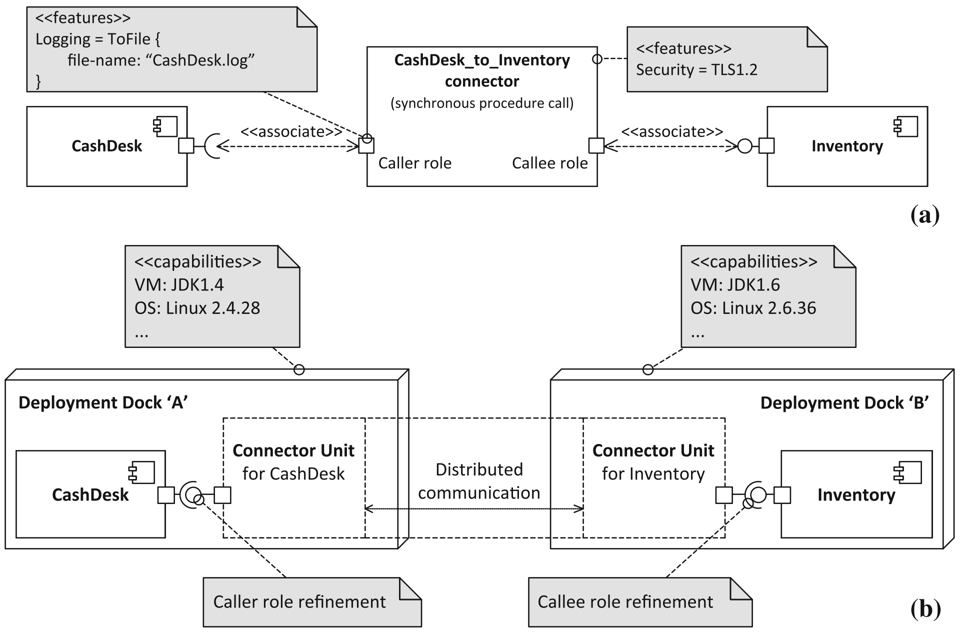


Fig. 3 Example of the application designer perspective: **a** requirements view, **b** deployment view

deployment docks. Deployment dock capabilities are key properties of the execution environment, driving selections of middleware technology for remote communication and are based on the OMG D&C standard [33]. For example, the capabilities of the dock "A" indicate the availability of Java virtual machine version JDK 1.4 and underlying Linux 2.4.28. Thus, the unit for the `CashDesk` component has to be able to run and communicate in such runtime environment.

2.2 Connector designer perspective

From the connector designer perspective, the design view of a connector focuses on describing the connector implementation by composition of (hierarchical) *elements*.

An important idea is that the individual parts of a connector implementation are designed in advance, anticipated for reuse. To allow this, the connector role in a particular application (i.e., connecting specific components while considering their deployment) is abstracted away by parameterizing the design-view concepts. Namely, the parameters are the communication style, feature requirements, and capabilities, being thus the only binding concepts of the application-designer and connector-designer perspectives.

At the top level, the structure of a connector is described as a *distribution architecture* (Fig. 4a), defining units, their interconnections, and a refinement of their requirements. This refinement addresses the communication style by means of assigning roles to units and also addresses the NFPs by delegating required features to units. Further, each unit is to be refined by an element. However, to keep the distribution architecture general and reusable, such an element is determined just by means of the following characteristics: (a) the black-box view of the element, the *element type*, and (b) its features (including the ones delegated to it as explained below). For example, the communication style in Fig. 4a is procedure call and thus the unit `For_Callee` is assigned the role `Callee`. Here, the `Security` feature requirement is delegated to the `For_Callee` unit, implementation of which is characterized by the `ServerSkeleton` element type.

The (potentially) *composite* elements are further recursively refined by a composition of sub-elements up to *primitive* elements (Fig. 4b). For example, the composite element `LoggedClientStub` refining the unit `For_Caller` is further refined by a composition of `FileLogger` and `RMIStub` primitive elements.

Elements interact via *ports* (Fig. 4b). A port can be either local or remote. A *local* port (e.g., of `FileLogger`) serves for internal communication of elements not directly participating in the communication among units. Since communication via local ports is based on local procedure calls, a local port is specified either as provided or required to emphasize where the communication is initiated. This also implies that any role is implemented as a local port. A *remote* port (e.g., of `RMIStub`) serves for (potentially) distributed communication among units. In this case, the actual form of communication depends on the employed middleware and thus the specifics of the communication are intentionally abstracted at the level of unit (and corresponding element) specification; in particular, there is no explicit distinction between provided/required remote ports.

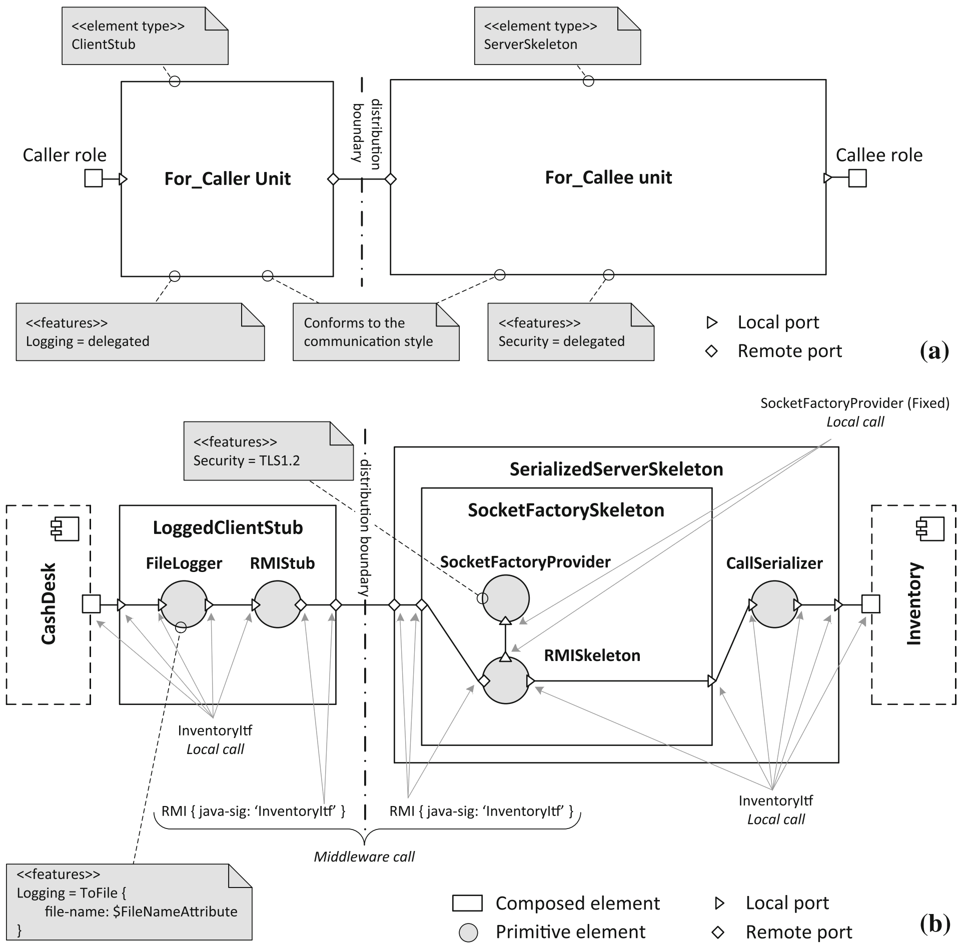


Fig. 4 Example of the connector designer perspective: **a** distribution architecture, **b** a full composition of a connector

To capture the element hierarchy at design time, each element is refined as a gray box by an *element architecture* refining the element's type by specifying the horizontal composition of sub-elements. Here, a sub-element is characterized just by its element type (defining its ports) and its features. Each element architecture refining an element type has also to address the required features either (i) directly, or (ii) by delegating them further to its sub-elements. This facilitates automated synthesis and reuse of hierarchically composable elements in a way similar to hierarchical components [7,11,38]. The element architecture also captures the port bindings, including both bindings among sub-elements and port delegation (illustrated below). For example, the element architecture of the `SocketFactorySkeleton` element in Fig. 4b defines two sub-elements: `SocketFactoryProvider` and `RMISkeleton`. Here, according to the element architecture, `SocketFactoryProvider` is required to provide a single local port (as defined by its element type, for brevity not captured in the figure), as well as to address the `Security` feature (delegated here, being addressed directly by the `SocketFactoryProvider` with the value `TSL1.2`). The element architecture also defines a binding between the sub-components (via the local port of the `SocketFactoryProvider` and one of the local ports of `RMISkeleton`), and the delegation of the remote port of `SocketFactorySkeleton` to the remote port of the `RMISkeleton` sub-element. The exact definitions of the element architectures and element types of the elements in Fig. 4b are in Sect. 4.1.

Eventually, every port is associated with a *signature* (Fig. 4b), being in principle the interface type of the port. In a regular case, the signatures are inferred from the interface types associated with role instances. These types are propagated through the element hierarchy via element bindings (forming a call chain, e.g., `InventoryItf` determines the signatures of `FileLogger` and other elements on the call path up to `CallSerializer`). In a special case the signature is explicitly defined (e.g., the port signature of `SocketFactoryProvider`).

The process of signature propagation is subject to signature constraints, which determine the relation among signatures of one element. For example, they may establish the equality of signatures of provided and required ports—as in case of `FileLogger`, or they may express how the signature of a provided port has to be modified to become a signature of a remote port—as in the case of `RMIStub`.

As for features, in general, a part of a feature value may be left unspecified, to be later on determined from requirements view by feature delegation. This is captured by the concept of element *attribute*, serving to parameterize features (e.g., the `file-name` attribute of `FileLogger` parameterizes the `Logging` feature).

In order to achieve abstraction of a particular deployment, each element architecture contains its *runtime-environment requirements*, expressed as constraints on dock capabilities.

2.3 Summary of the concepts and problem refinement

To summarize, a particular connector instance is represented as a fully fledged hierarchical composition of elements (involving their element architectures, refinement of sub-elements, association of signatures to ports, and assignment of element attributes), divided into units according to the required deployment, while addressing the required features. For the purpose of this paper, we describe such a connector instance by a meta-model (Fig. 5), expressed in terms of sets and relations. Note that the meta-model does not capture the design-only concepts such as element type, communication style, and dock capabilities. Instead, they are already "blended" in the other meta-model concepts for brevity (e.g., the element type is blended in the element architecture).

In the rest of this paper, the representation of a particular connector instance (and thus an instance of the meta-model) will be referred to as a CIC.

A particular CIC is determined by both (a) the requirements and deployment view, and (b) the design view. Nevertheless, given both (a) and (b), while some of the meta-model sets and relations forming the CIC are fully determined (solid line), some are underspecified (dashed line), and therefore determined only partially (*alterable sets and relations*). In other words, multiple realizations of the alterable sets and relations can satisfy the given (a) and (b), i.e., multiple variants of the CIC exist (for example, the variants can be introduced by the existence of several element architectures suitable for refining a single sub-element). Note that while (a) is connector-specific, (b) can be predefined and shared among multiple connectors.

The level of underspecification of the alterable sets and relations can be described in terms of constraints inferred from (a) and (b). Some of these constraints are general for all (a) and (b), e.g., an `Element` uses only those

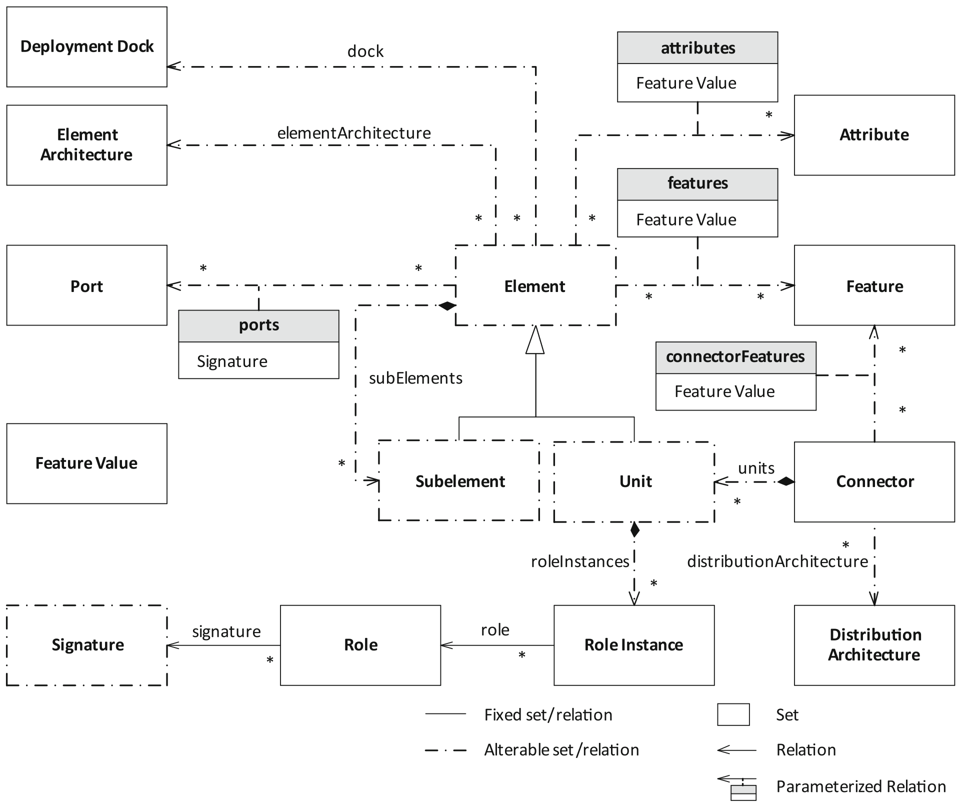


Fig. 5 Sets and relations describing a connector: meta-model

`ports`, which are defined by the element type refined by its `elementArchitecture`, while some are specific, e.g., an `Element` is deployed only on a `dock`, capabilities of which are compatible with its runtime-environment requirements. For the sake of brevity, we do not explicitly represent the general constraints in the meta-model.

Overall, the problem of finding a CIC can be interpreted as finding a realization of the alterable sets and relations with respect to (a) and (b). If multiple alternatives exist, an optimal one should be chosen (a discussion on optimization criteria is in Sect. 6.3). Specifically, such a realization of the alterable sets and relations in CIC embodies the following:

- a particular distribution architecture that conforms to the communication style specified in the requirements specification and definition of connector units,
- vertical composition of element architectures (each of them describing a horizontal composition of its sub-elements) determining, which element architecture to choose for each (sub-) element in a distribution architecture (recursively at all levels of element nesting),
- actual parameters for the elements by providing actual signatures for the elements' ports and determining values of the attributes.

However, depending on the (a) and (b), the number and complexity of the constraints determining the alterable sets and relations, as well as the actual size of the sets/relations, can grow too much to be feasible in a "manual" way. Therefore, there is a need for an automated method—ARCAS.

Technically, the requirement view of a connector is assumed to be described by a *requirements specification*, the deployment view by a *deployment specification*, and all the concepts featured in the design view by an *artifact specification*.

3 Overview of the ARCAS method

Following the ideas of Sect. 2.3, automated resolution of CIC can be viewed as a relational constraint-solving problem. Here, the constraint specification is formed by the realization of the fixed sets and relations, as well as the constraints over the alterable sets and relations. A realization of the alterable sets and relations represents a solution to the problem. For this purpose, it is advantageous to employ a constraint-solving technique based on a modeling language rich enough to express the required concepts. In general, relational constraint solving languages such as relational logic are well suited to this purpose.

Informally put, we create a constraint specification, referred to as a *connector theory* (CT)—in the sense of logic, capturing specification of a particular connector in terms of a logic theory, so that a model of such a theory represents a CIC of the specified connector. More precisely, since a CT may have more than one model, this theory basically represents a description of a set of all the alternative CICs of the specified connector. In other words, a model of a CT (in the sense of logic) provides a representation of all the sets and relations of the CIC meta-model from Fig. 5; thus, Fig. 5 also represents the meta-model of the CT.

Specifically, a CT consists of four parts (Fig. 6) based on the specifications determining a connector: (i) a definition of the abstractions global to all CT theories (reflecting the meta-model), (ii) images of candidate element architectures, (iii) images of candidate distribution architectures, (iv) an image of the requirements imposed by requirements and deployment specifications.

Here, an "image" means a projection of a particular specification artifact into the language of CT (e.g., projection of a particular element architecture).

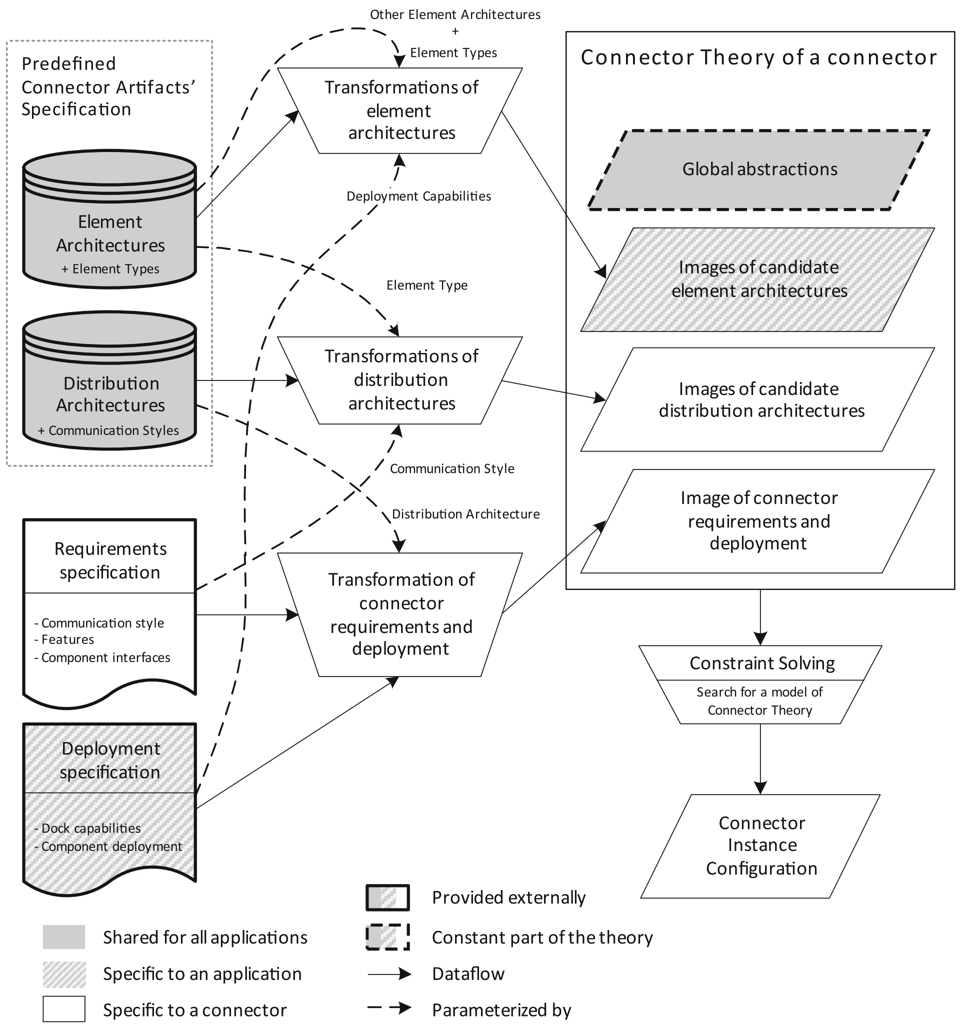
Semantically, a model of a CT represents (a) a correct CIC (the elements in the CIC are able to cooperate), and (b) a desired CIC (the corresponding connector complies with the requirements and deployment specifications).

To keep the CT simple and feasible for CIC resolution, the representation of communication styles, element types, and deployment is subject to "inlining" (substitution). For example, in (ii) and (iii) each element type is inlined by a list of all the element architectures suitable for this type.

It should be emphasized that the parts (ii) and (iii) consider only a subset of the available element architectures, resp., distribution architectures. In the former case, only the element architectures able to run in the deployment docks specified by the deployment specification are considered, in the latter case, only the distribution architectures conforming to the specified communication style are included.

Advantageously, except for part (i), which is shared for all applications and created by hand in advance, a CT can be constructed in an automated way by transformation of the specifications into formulas of the selected CT language. In ARCAS, for constructing each of the parts (ii)–(iv), there is a specific transformation (Fig. 6). A transformation processes a corresponding specification while considering several other specifications as additional parameters, producing the corresponding specification's image. For example, the transformation of a single element architecture specification considers the specifications of other element architectures, element types, and deployment capabilities as additional parameters. A particular CT is constructed by applying transformations to all the relevant specifications.

Once a CT is constructed, a constraint solver available for the selected CT language is employed to find out a model of the CT. This model can be easily programmatically converted to the corresponding CIC (Fig. 6).



Other Element Architectures + Element Types
Predefined Connector Artifacts' Specification
Element Architectures + Element Types
Distribution Architectures + Communication Styles
Transformations of element architectures
Deployment Capabilities
Element Type
Transformations of distribution architectures
Communication Style
Distribution Architecture
Transformation of connector requirements and deployment
Requirements specification
- Communication style
- Features
- Component interfaces
Deployment specification
- Dock capabilities
- Component deployment
Connector Theory of a connector
Global abstractions
Images of candidate element architectures
Images of candidate distribution architectures
Image of connector requirements and deployment
Constraint Solving
Search for a model of Connector Theory
Connector Instance Configuration
Provided externally
Constant part of the theory
Dataflow
Parameterized by
Shared for all applications
Specific to an application
Specific to a connector

Fig. 6 Overview of the ARCAS method

The specifications are described in more detail in Sect. 4. The transformations of the specifications producing a particular CT are elaborated in Sect. 5. Since the Alloy modeling language [22–24] and its solver Alloy Analyzer are good candidates for representing CT (Alloy provides convenient syntax for definition of relations and their constraints), in Sect. 6 we describe the representation of CT using Alloy.

Referring back to Sect. 1 it should be emphasized that ARCAS is intended to be applied at either the deployment or runtime stage of an application. In the latter case, this would be due to a runtime modification of the architecture and/or deployment of the application.

4 ARCAS input: specifications

In this section, we will elaborate on the specifications required as the input of the transformations in ARCAS (Fig. 6); the specifications were conceptually outlined in Sect. 2. In principle, each of them is defined by its metamodel, i.e., abstract syntax, and for practical reasons ARCAS includes connector definition language (CDL), i.e., concrete syntax, in which we will provide examples.

We will fully describe the abstract syntax and semantics, and give examples of the element type and element architecture specifications. For brevity, the other specifications (i.e., communication style, distribution architecture, requirements, and deployment specifications) are illustrated by an example based on Figs. 3 and 4, while their abstract syntax is provided in [26].

4.1 Element type and element architecture specifications

The basic abstraction of `ElementArchitecture` (Fig. 7) is `ElementType`, which defines the external interface of an element, i.e., its ports. A port is defined as provided, required, or remote. The following illustrates specification of

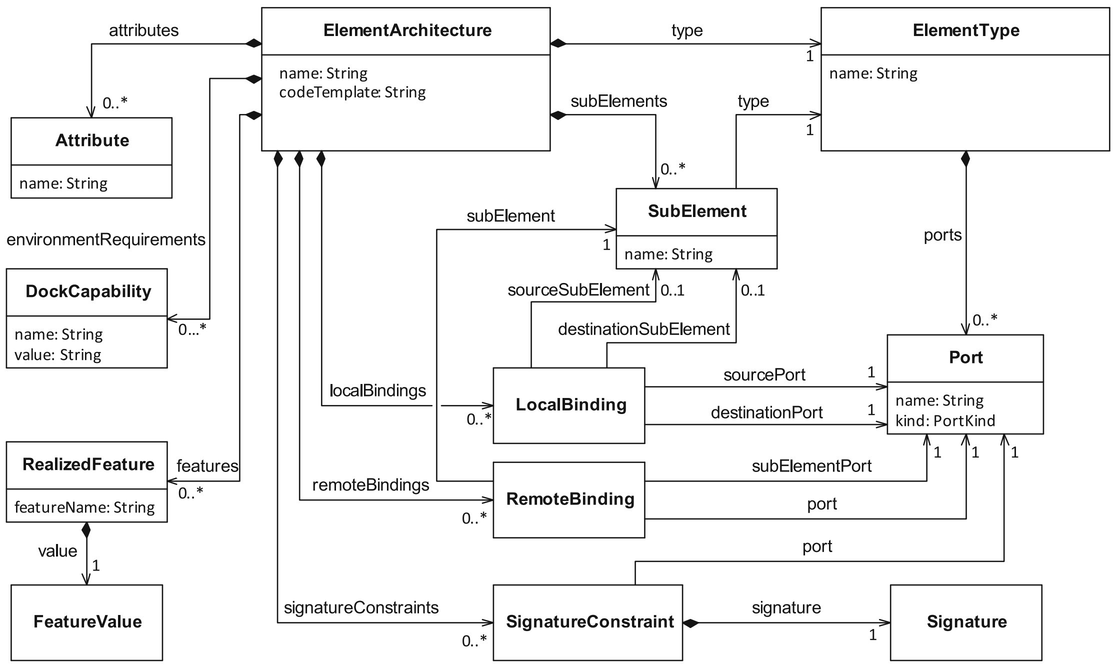


Fig. 7 Abstract syntax of element architecture specification

`ClientStub`, `SocketBasedSkeleton`, and `Logger` element types (the types of `LoggedClientStub`, `RMISKeleton`, and `FileLogger` from Fig. 4b) in CDL:

```
element-type ClientStub
    local-provides: callIn          -- <port kind>: <port name>
    remote: mw

element-type SocketBasedSkeleton         element-type Logger
    local-requires: factory                  local-provides: callIn
    local-requires: callOut                  local-requires: callOut
    remote: mw
```

An element architecture refines an element type by determining a hierarchical composition of elements at two adjacent levels of nesting (the lower level is determined by the `subElements` relation in Fig. 7). Thus, only horizontal composition (of the elements at the lower level) is specified for `ElementArchitecture`; as to vertical composition, it is here expressed by enforcing a particular element type for a sub-element.

Further, indication of port bindings is defined separately for pairs of provided and required ports (`LocalBinding`), and for delegation among remote ports (`RemoteBinding`). The former case expresses either delegation or communication at the same level of nesting.

Port signature propagation is captured by expressing relations between the ports of a single element architecture (`SignatureConstraint`); these relations enforce equal or equally parameterized signatures of the related ports. This ensures an abstraction over element implementation, providing only the information important for vertical composition. The way `Signature` is defined in Fig. 8a indicates that a signature takes the form of either a signature variable, acting as a signature parameter, or a signature term (possibly parameterized).

As for the features (Fig. 8b) realized by the element architecture (`RealizedFeature`), a feature value is represented by a possibly structured feature term. The value of a feature (sub-)term is either delegated to a sub-element (`FeatureDelegation`), explicit (set explicitly via a fixed feature term), or parameterized by an *attribute* (`AttributeMapping`), playing a similar role as signature variables in signature terms. Assuming an element *e*, the actual value of its attribute is defined either in a parent element (recursively) or directly at the level of the whole connector (in the requirements specification); in both cases, the value is propagated to *e* by feature delegation. For delegating a feature value, the higher-level element architecture needs to anticipate the features of its sub-elements. However, in a sub-element architecture corresponding features are not strictly required; if they are missing, this implies ignoring the delegated feature in the parent element.

For simplicity, environment requirements are represented by means of name-value pairs (DockCapability).

The CDL specification of the LoggedClientStub, FileLogger, and RMISkeleton element architectures from Fig. 9 (elaboration from Fig. 4b) can take the following form:

```
element-architecture LoggedClientStub
    of-type: ClientStub
    sub-elements:
        stub: ClientStub                 -- <name>: <type>
        logger: Logger
    bindings:
        this.callIn -> logger.callIn     -- -> local binding
        logger.callOut -> stub.callIn    -- <subelement>.<port>
        this.mw <-> stub.mw              -- <-> remote binding
    features:
        logging -> logger.logging        -- feature delegation

element-architecture FileLogger
    of-type: Logger
    signature-propagation:
        callIn: $I                       -- $I is a signature variable
        callOut: $I
    features:
        logging: ToFile{              -- explicit feature, set to ToFile{…}
                    name: $FileName   -- ‘name’ is a feature term parameter
                 }                    -- $FileName is an attribute
    environment-requirements: {“Java VM” -> “JDK 1.4” }

element-architecture RMISkeleton
    of-type: SocketBasedSkeleton
    signature-propagation:
        callOut: $I                   -- $I is a signature variable
        mw: RMI{ java-sig: $I }       -- parameterized signature
                                      -- ‘java-sig’ is a signature parameter
        factory: FactorySignature     -- fixed port signature
    environment-requirements: {“Java VM” -> “JDK 1.6” }
```

Here, an important part is the definition of features and attributes, element types, signature propagation, and environment requirements.

FileLogger and logging give an example of an explicit feature. The feature value is parameterized by the FileName attribute, which is used in the implementation of FileLogger. The logging feature of LoggedClientStub illustrates the delegation of features. Here, the meaning is that LoggedClientStub provides logging only if this feature is provided by its logger sub-element (in the positive case, logger also defines the value of logging).

As to element types, notice that both LoggedClient Stub and its sub-element stub are of the same element type; recall RMIStub element architecture assigned to stub in Fig. 4b. Thus, both LoggedClientStub and RMIStub can be used to refine the For_Caller unit. In this example, LoggedClientStub was preferred over RMIStub because of its logging feature.

Signature propagation is illustrated by FileLogger, where the use of the same signature variable $I for both callIn and callOut indicates that both ports have to have the same signature. A more complex example is RMISkeleton, where the signature of the callOut port is propagated as the java-sig parameter of the mw port's signature. The RMISkeleton specification also illustrates definition of a fixed port signature FactorySignature.

The environment requirements are expressed by means of required dock capability values. RMISkeleton requires the target deployment dock to support JDK 1.6.

4.2 Remaining specifications

4.2.1 Communication style

As outlined in Sect. 2, specification of a communication style determines the connector roles and their cardinality.

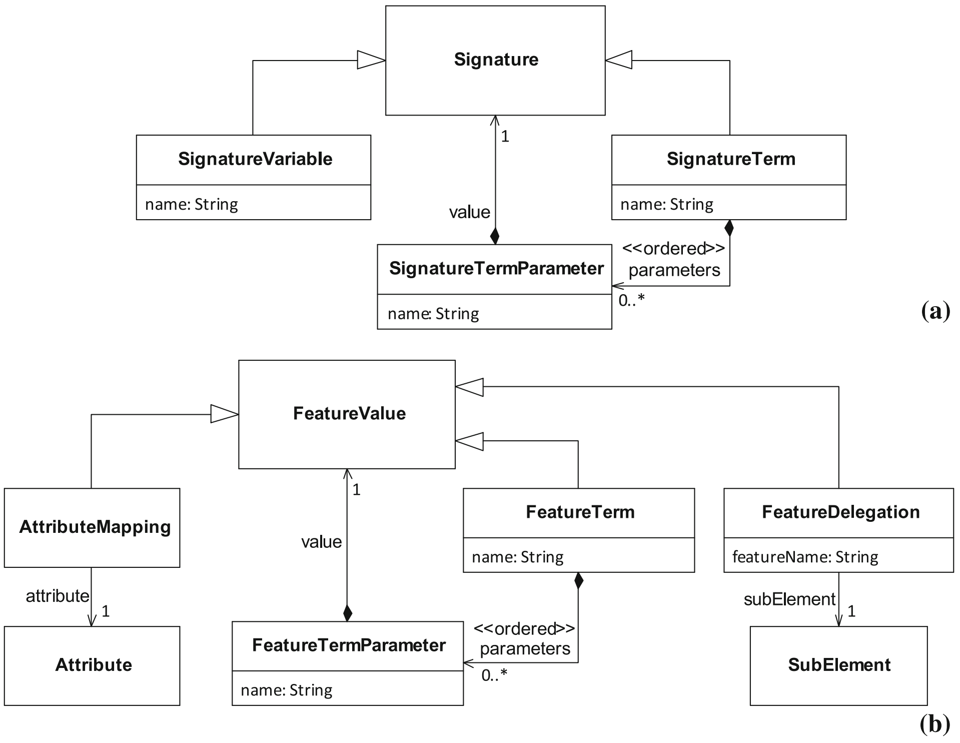


Fig. 8 Abstract syntax for specification of **a** signatures, **b** features

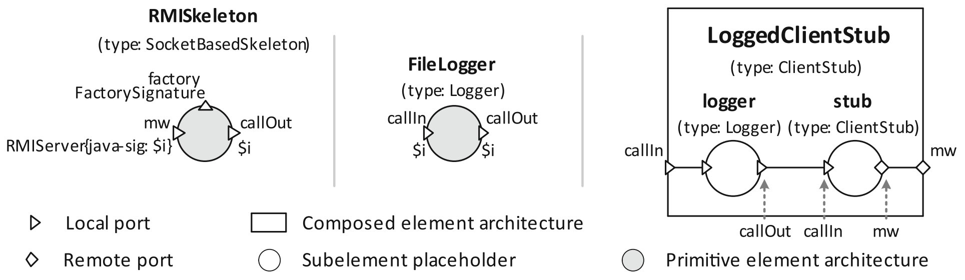


Fig. 9 Elaboration of FileLogger, RMISkeleton, and LoggedClientStub element architectures from Fig. 4b

The communication style of the connector from Fig. 3a can be specified as:

```
communication-style ProcedureCall
    roles:
        Callee              -- lower bound = 1, upper bound = 1
        Caller(0..n)        -- lower bound = 0, upper bound = n
```

4.2.2 Requirements

The key idea of requirements specification is that the particular connector architecture to be used for the connector is not specified explicitly; instead, it is declaratively determined by the selected communication style, required features, and desired deployment of components.

As the requirements specification is application-specific, the components to be connected and their interfaces (including signatures) have to be defined. Based on this, requirements are specified by selecting a communication style, mapping roles to the component interfaces (i.e., defining connector endpoints), and by defining the required features of the connector. Feature requirements are determined by enumerating the acceptable feature values, where a value is represented by a feature term. There is also the option to define a feature requirement for a particular endpoint. Feature requirements can be composed using propositional operators.

The specification of requirements for the example from Fig. 3a can take the following form:

```
component CashDesk
    requires: InventoryItf inventory -- <interface kind>: <signature> <name>

component Inventory
    provides: InventoryItf inventory

connector CashDesk_to_Inventory
    communication-style: ProcedureCall
    endpoints:
        CashDesk.inventory as Caller -- <component.interface> as <role>
        Inventory.inventory as Callee
    features:
        security in { TLS1.2, SSL3 } -- <feature> in {<possible values>}
        Inventory.inventory.logging in { ToFile { name: “inventory.log” }}
                                   -- <component>.<interface>.<feature>
                                   -- ‘name’is feature term parameter
```

4.2.3 Deployment

A deployment specification expresses the desired deployment of the connected components (not of connector deployment!). It determines the allocation of components to deployment docks and the capabilities of the deployment docks. The dock capabilities are based on the OMG D&C [3,33] standard (syntactically expressed by means of typed name-value pairs). The specification of deployment for the example from Fig. 3b can take the following form:

```
allocation
    CashDesk to DockA
    Inventory to DockB

dock DockA
    capabilities: {“Java VM” -> “JDK 1.4”, “OS” -> “Linux 2.4.28”}

dock DockB
    capabilities: {“Java VM” -> “JDK 1.6”, “OS” -> “Linux 2.6.36”}
```

4.2.4 Distribution architecture

The distribution architecture specification determines the desired communication style, the element type of the top-level element architectures refining units, their cardinality, the units’ remote bindings, and mapping of roles to ports in units. The specification also explicitly states how features are addressed by specific units (feature delegation). The distribution architecture specification of the connector from Fig. 4a can take the following form:

```
distribution-architecture RPC
    communication-style: ProcedureCall
    units:
        For_Callee: ServerSkeleton         -- <name>: <element type>
        For_Caller (0..n): ClientStub      -- <name>(<cardinality>): <type>
    remote-bindings:
        For_Caller(i).mw <-> For_Callee.mw -- <unit>.<port>
    role-mapping:
        For_Caller(i).callIn as Caller(i)  -- <unit>.<port> as <role>
        For_Callee.callOut as Callee
    features:
        security -> For_Callee.security    -- feature delegation
```

An important concept is the specification of multiplicity of unit instances such as `For_Caller` (θ..n): ClientStub, which says that there will be up to n `For_Caller` units of the type `ClientStub`. In consequence, multiplicity is to be expressed also in `remote-bindings` and `role-mapping`. For example, `For_Caller(i).mw` $<->$ `For_Callee.mw` means that the `mw` port of all the `For_Caller` units is bound to the mw port of the `For_Callee` unit. Likewise, `For_Caller(i).callIn` as `Caller(i)` means that the `callIn` port of each `For_Caller` unit has the role `Caller`.

5 Constructing connector theory by transformations

As outlined in Sect. 3, a connector theory can be constructed in an automated way by transformations of the specifications. In this section, we describe the transformations and their products in terms of propositional logic with relational

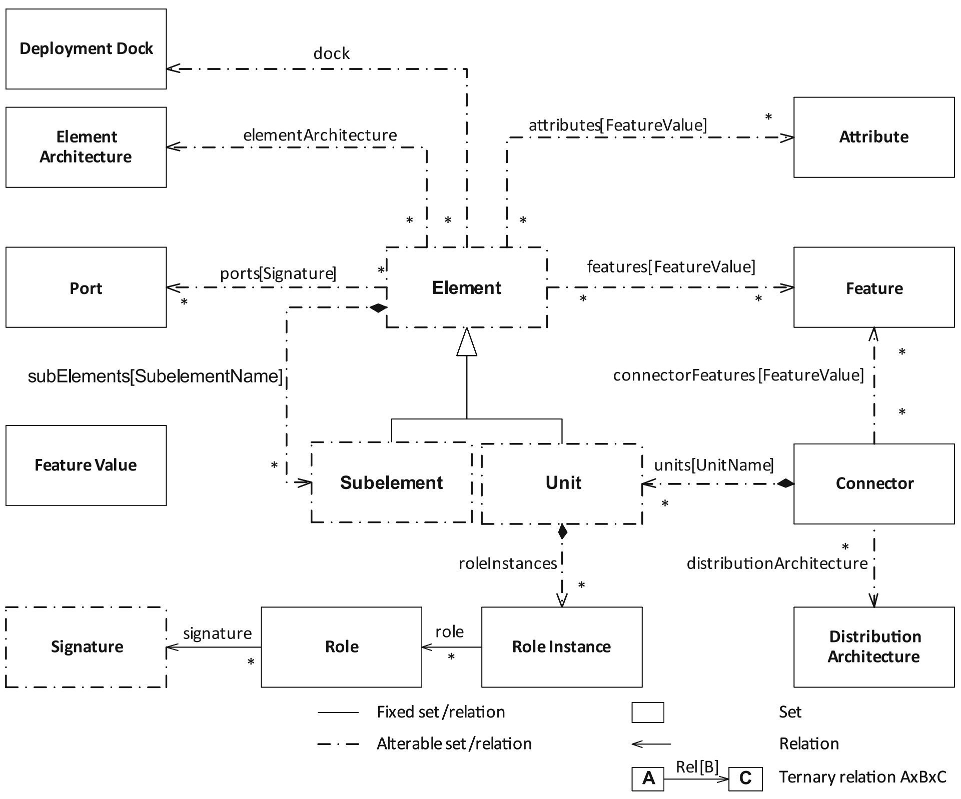


Fig. 10 Modified connector theory meta-model (by ternary relations)

calculus, using only basic logical and relational operators. In general, a transformation results in a formula. The set of all formulas resulting from transformations of the specifications relevant to a particular connector forms the corresponding connector theory.

The formulas express the constraints on alterable sets/relations by binding them to the fixed ones (Fig. 5). In the formulas, we rely on predicates, the semantics of which is explained informally in this section to help the reader get an intuitive understanding of them; detailed semantics is provided only for a selection of the predicates. We further elaborate on the description of selected predicates in Sect. 6 by implementing them in Alloy.[1] There, we also illustrate a construction of the fixed sets and relations. For the fixed sets and relations, we assume that a realization is available, implied by the specifications (both those of predefined artifacts, and those of requirements and deployment, specific to a connector).

For defining the actual transformations based on predicate logic with relational algebra, it is advantageous to introduce a slightly modified version (Fig. 10) of the connector meta-model from Fig. 5. The rationale for such modification is that certain relations of the original meta-model (i.e., *features, connectorFeatures*, *attributes*, and *ports*) are parameterized. In order to capture this in relational calculus, the originally binary parameterized relation is now expressed as a ternary relation (e.g., the relation *features* in Fig. 10). In addition, given an element *e*, in order to express constraints over its particular sub-element *se*, the name of the sub-element is explicitly employed. This is necessary, since the constraints on *se* are imposed in the element architecture of *e*. Therefore, the original binary relation *subElements* is replaced by a ternary relation introducing an identifier of the sub-element (the identifier is taken over from the definition of *se* in the element architecture of *e*). The relation *units* is to be modified in a similar way to *subElements*.

Because the formulas resulting from a single transformation have a fixed internal structure, we present the transformations in terms of parameterized templates (Fig. 11). Syntactically, the template contains a number of parameter placeholders (e.g., ⟨`se`⟩ in Fig. 13) and specific constructs (e.g., ⟨`foreach`⟩ in Fig. 13), determining the way the template parameters are projected into the template text. The template parameters are derived from the transformation parameters (Sect. 3). Based on actual transformation parameters, the transformation produces a specification's

[1] The full formalization in Alloy is provided at http://d3s.mff.cuni.cz/projects/components_and_services/arcas/.

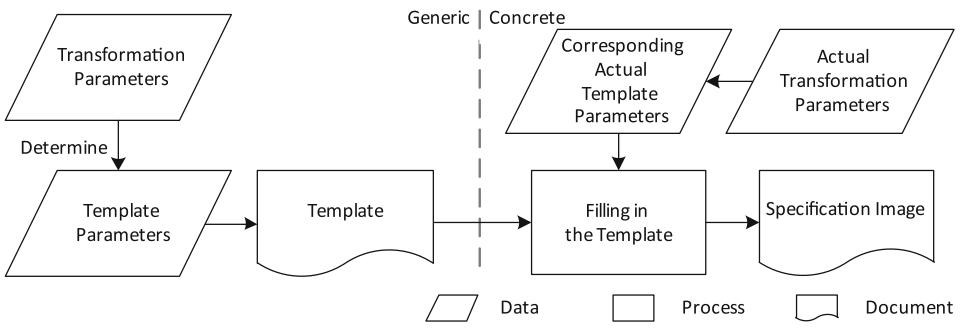


Fig. 11 Generic workflow of transformations

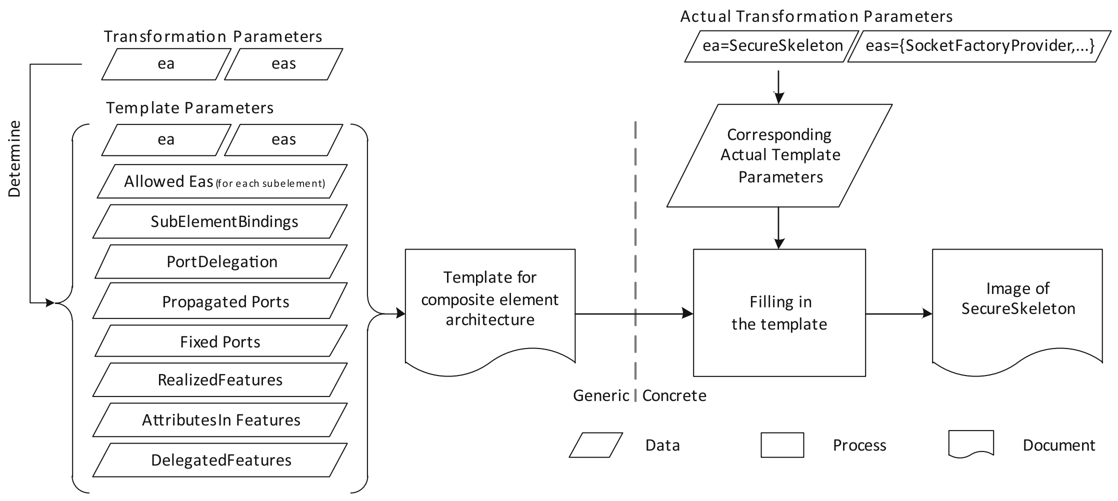


Fig. 12 Transformation workflow for composite element architecture

image (i.e., a part of the connector theory under construction) by filling in the template.

In detail, we describe the transformation of a composite element architecture specification focusing on the corresponding template and derivation of the template parameters. For brevity, the effect of other transformations (i.e., of primitive element architecture, distribution architecture, and requirements specifications) is illustrated by an example of their product, while their full description is provided in [26].

5.1 Transforming specification of a composite element architecture

The transformation workflow for a composite element architecture specification is illustrated in Fig. 12.

5.1.1 Template

The template (Fig. 13) formulates the constraints on any element *e* employing a particular composite element architecture *ea*, the specification of which is subject to the transformation. The constraints are derived from the semantics of the composite element architecture concept (Sects. 2 and 4.1). The template takes the form of an implication.

Technically, in the meta-model these constraints restrict the relations that are (potentially transitively) related to *e* as an element of the set *Element*. The antecedent of the implication assumes validity of the *ElementHasEA* predicate over the *elementArchitecture* relation (*e* employs *ea*). The consequent of the implication includes (i) general element architecture constraints, (ii) sub-element constraints, (iii) port constraints, (iv) signature constraints, and (v) feature constraints.

As to (i), after enforcing that *e* contains the ports defined by the element type of *ea* (the *ports* relation is constrained by *AvailablePorts*), the set of sub-elements of *e* is constrained to make it conform to *ea* (via constraining the *subElements* relation by *SubElements*). In addition, it is desirable to make sure that only the realized features and declared attributes of *ea* may be employed by *e* (the *features* and *attributes* relations are constrained by *SupportedFeatures*, resp., *RequiredAttributes*).

As to (ii), the definitions of the sub-elements are reflected as follows. For each sub-element, the set of element architectures that can be employed for its refinement is explicitly

```
$\forall e \in Elements : ElementHasEA(e,$ <ea>$) \implies ($
    $AvailablePorts(e,$ <ea.type.ports>$) \wedge$
    $SubElements(e,$ <ea.subElements>$) \wedge$
    $SupportedFeatures(e,$ <ea.features>$) \wedge$
    $RequiredAttributes(e,$ <ea.attributes>$) \wedge$

    -- definition of sub-elements
    <foreach se in ea.subElements >
        -- <se> sub-element
        $AllowedEAsForSubElement(e,$ <se>, <AllowedEAs$_{se}$>$) \wedge$
        $PropagateDeployment(e,$ <se>$) \wedge$
    <endforeach>

    -- port bindings
    <foreach (se1, port1, se2, port2) in SubElementBindings >
        $PortBinding(e,$ <se1>, <port1>, <se2>, <port2>$) \wedge$
    <endforeach>
    -- port delegation
    <foreach (port, se, seport) in PortDelegations>
        $PortDelegation(e,$ <port>, <se>, <seport>$) \wedge$
    <endforeach>

    -- signature propagation
    <foreach (port1, port2) in PropagatedPorts >
        $PropagateSignature(e,$ <port1>, <port2>$) \wedge$
    <endforeach>
    -- enforcing fixed signatures
    <foreach (port1, port2) in FixedPorts >
        $PortHasSignature(e,$ <port>, <signature>$) \wedge$
    <endforeach>

    -- enforcing values of the directly realized features
    <foreach f in RealizedFeatures >
        $FeatureHasValue(e,$ <f.name>, <Encode(f.value)>$) \wedge$
    <endforeach>
    -- defining attributes referred in feature values
    <foreach (feature,parameter,attribute) in AttributesInFeatures>
        $FeatureValueUsesAttribute( e,$ <feature>, <parameter>, <attribute>$) \wedge$
    <endforeach>)
    -- feature delegation
    <foreach (feature, se, sefeature) in DelegatedFeatures >
        $FeatureDelegation(e,$ <feature>, <se>, <sefeature>$) \wedge$
    <endforeach>
)
```

Fig. 13 Template for image of composite element architecture

prescribed (the *subElements* and *elementArchitecture* relations are constrained by *AllowedEAsForSubElement*). This ensures the refinement consistency of the sub-elements. Note that the sub-element's ports are determined by the prescribed element architectures (as they have the same element type). Since *e* has to be deployed on the same node as its sub-elements, their deployment is propagated to *e* (*subElements* and *node* are constrained by *PropagateDeployment*).

As to (iii), the port bindings and port delegations are reflected by forcing the signatures of the involved ports to be equal (via constraining the *subEelements* and *ports* relations by *PortBinding*, resp., *PortDelegation*). This, along with enforcing signature propagation in the images for sub-element architectures, ensures the composition consistency of the sub-elements.

As to (iv), according to the signature propagation constraints for *ea*, the signatures/signature parameters of the involved ports of *e* have to be equal (*ports* is constrained by *PropagateSignature*). Note that we have abstracted away details of signature propagation for the sake of brevity. It is also necessary to enforce the fixed port signatures (*ports* is constrained by *PortHasSignature*).

As to (v), the values of the feature directly realized by *ea* have to be reflected by *e* (via constraining *features* by *FeatureHasValue*). Since a feature value can use an attribute as its parameter, the attribute value and the respective feature

Table 1 Semantics of predicates for composite element architecture image

Predicate name	Description	Formula
AvailablePorts (e, portSet)	The element *e* contains exactly the ports in *portSet*	$\forall p \in Port, \forall s \in Signature : (e, s, p) \in ports \Leftrightarrow p \in portSet$
AllowedEAsFor-SubElement (e, subelement, eas)	The sub-element *se* (of the element *e*), having the name *subelement*, can refine only an element architecture from *eas*	$\forall se \in Element : (e, subelement, se) \in subElements \Rightarrow (\forall ea \in ElementArchitecture : (se, ea) \in elementArchitecture \Rightarrow ea \in eas)$
PortBinding (e, subelement1, port1, subelement2, port2)	For the sub-elements *se1* and *se2* of *e* with names *subelement1*, resp., *subelement2*, the *port1* of *se1* and *port2* of *se2* have the same signature	$\forall se1, se2 \in Element, (e, subelement1, se1) \in subElements \wedge (e, subelement2, se2) \in subElements \Rightarrow (\forall s \in Signature : (se1, s, port1) \in ports \Leftrightarrow (se2, s, port2) \in ports)$
FeatureValueUses-Attribute (e, feature, parameter, attribute)	For the feature value (*fv*) of *feature* in the element *e*,*attribute* of *e* and *parameter* of *fv* have the same value	$\forall fv \in FeatureValue, \forall av \in FeatureValue : \{(e, fv, feature) \in features \wedge (e, av, attribute) \in attributes\} \Rightarrow (fv, av) \in parameter$
FeatureDelegation (e, feature, subelement, se-feature)	For the sub-element *se* (of the element *e*) with name *subelement*, *feature* of *e* and *se-feature* of *se* have the same value	$\forall se \in Element : (e, subelement, se) \in subElements \Rightarrow (\forall fv \in FeatureValue : (e, \nu fv, \epsilon feature) \in features \Leftrightarrow (se, fv, se - feature) \in features)$

parameter value have to be equal (*features* and *attributes* are constrained by *FeatureValueUsesAttribute*). Finally, the delegated features have to be captured (*subElements* and *features* are constrained by *FeatureDelegation*).

For illustration, the semantics of "representative" predicates is presented in Table 1 via a textual description and an equivalent logical formula.

5.1.2 Transformation parameters

The transformation parameters are: (i) the element architecture *ea* to be transformed and (ii) the set of all predefined element architectures *eas* (Fig. 12).

5.1.3 Template parameters

In addition to the element architecture *ea* to be transformed and the set of all predefined element architectures *eas*, these are $AllowedEAs_{se}$ (separately for each sub-element of *ea*), *SubElementBindings, PortDelegation, RealizedFeatures, AttributesInFeatures*, and *DelegatedFeatures* (Fig. 12).

$AllowedEAs_{se}$ contains the element architectures that may be used for refining the sub-element *se* (decision is based on the element type of *se* defined by *ea*). *SubElementBindings* comprises all pairs of sub-elements of *ea* and their ports for which *ea* defines a binding. *PortDelegation* contains the tuples (ea port *port*, sub-element *se*, sub-element port *seport*) that participate in port delegation (including delegation of both local and remote ports). *RealizedFeatures* includes all the features realized directly by *ea*. *AttributesInFeatures* contains all the tuples *(feature, feature parameter, attribute)* such that *attribute* is used as the value of *feature parameter* which is part of the feature term associated with *feature*. *DelegatedFeatures* contains all the tuples (ea feature f, sub-element *se*, sub-element feature *sef*) such that f is mapped to *sef*.

Mathematically, the template parameters are determined from transformation parameters as shown in Fig. 14.

5.1.4 Example: image of SocketFactorySkeleton

Referring back to the example from Sects. 2 and 4, consider the actual transformation parameters $ea = SecureSkeleton$ and $eas = \{SerializedServerSkeleton, SocketFactorySkeleton, SocketFactoryProvider, RMISkeleton, \ldots\}$; the transformation will produce the image presented in Fig. 15.

5.2 Transforming specification of a primitive element architecture

To illustrate this transformation, we provide an example of the *FileLogger* primitive element architecture image (Fig. 16); a more detailed description of the template, as well as a description of the transformation and template parameters, is given in [26].

$$\begin{aligned}
AllowedEAs_{se} =& \{subea \mid subea \in eas \wedge subea.type = se.type\}\\
SubElementBindings =& \{(se1, port1, se2, port2) \mid se1, se2 \in ea.subElements \wedge\\
& port1 \in se1.type.ports \wedge port2 \in se2.type.ports \wedge\\
& \exists lb \in ea.localBindings \wedge\\
& IsSource(lb, se1, port1) \wedge IsDestination(lb, se2, port2)\}\\
PortDelegation =& \{(port, se, seport) \mid port \in ea.type.ports \wedge se \in ea.subElements \wedge\\
& se \in ea.subElements \wedge seport \in se.type.ports\\
& \wedge (\\
& \quad (\exists lb \in ea.localBindings \wedge IsSource(lb, ea, port) \wedge\\
& \quad IsDestination(lb, se, seport))\\
& \quad \vee\\
& \quad (\exists lb \in ea.localBindings \wedge IsSource(lb, se, seport) \wedge\\
& \quad IsDestination(lb, ea, port))\\
& \quad \vee\\
& \quad (\exists rb \in ea.remoteBindings \wedge IsEndpoint(lb, se, seport) \wedge\\
& \quad IsEndpoint(lb, ea, port))\\
&)\}\\
PropagatedPorts =& \{(port1, port2) | port1, port2 \in ea.type.ports \wedge\\
& sc1, sc2 \in ea.signatureConstraints \wedge\\
& sc1.port = port1 \wedge sc2.port = port2 \wedge\\
& UseTheSamePlaceholder(sc1.signature, sc2.signature)\}\\
FixedPorts =& \{(p, sc.signature) | p \in ea.type.ports \wedge\\
& sc \in ea.signatureConstraints \wedge\\
& sc.port = p \wedge IsFixedTerm(sc.signature)\}\\
RealizedFeatures =& \{f \mid f \in ea.features \wedge \neg IsFeatureMapping(f.value)\}\\
AttributesInFeatures =& \{(f.name, param.name, a.name | a \in ea.attributes \wedge\\
& f \in ea.features \wedge\\
& term \in TransitiveClosure(f.value) \wedge\\
& param \in term.parameters \wedge\\
& param.value = a\}\\
DelegatedFeatures =& \{(f, se, sef) \mid se \in ea.subElements \wedge f \in ea.features \wedge\\
& sef \in se.features \wedge IsFeatureMapping(f.value) \wedge\\
& f.value.featureName = sef.featureName \wedge\\
& f.value.subElement = se\}
\end{aligned}$$

Fig. 14 Template parameters for composite element architecture

5.2.1 Example: image of FileLogger

The image of *FileLogger* is very similar to an image of a composite element architecture (i.e., it formulates constraints on element *e* employing *FileLogger* in the form of an implication). Therefore, we will skip description of the common parts, referring the reader back to Sect. 5.1.

As for the constraints in the consequent, it is necessary to express that deployment of *e* is possible only to the docks compatible with *FileLogger* (*DockA*); this is reflected by constraining the relation *node* by *AllowedDeployment*. Further, *e* can feature only the ports defined by the *Logger* element type—the element type of *FileLogger* (*AvailablePorts*). It is also desirable to make sure that the set of sub-elements of *e* is empty since *FileLogger* is primitive (*SubElements*). As for the further constraints, only the realized features and declared attributes of *FileLogger* may be employed by *e* (*Supported-Features*, resp., *RequiredAttributes*).

Finally, after capturing the potential fixed signatures and signature propagation (*PropagateSignature*), and reflecting the values of the directly realized features (*FeatureHas-Value*), it is necessary to make the attributes and the particular feature values equal (*FeatureValueUsesAttribute*).

5.3 Transforming specification of a distribution architecture

To illustrate this transformation, we provide an example of the *RPC* distribution architecture (Fig. 17); a more detailed description of the template, as well as a description of the transformation and template parameters, is given in [26].

5.3.1 Example: image of RPC distribution architecture

Again, the image formulates constraints in the form of an implication (Fig. 17). The constraints relate to the whole connector, i.e., the image imposes the constraints on the current connector in case it employs the distribution architecture *RPC*. Note, that the corresponding element of the *Connector* set from the meta-model is not explicitly mentioned (in contrast to *e* in the case of element architecture), since we assume

```
∀e ∈ Elements : ElementHasEA(e, SocketFactorySkeleton) ⟹ (
    AvailablePorts(e, {Mw, CallOut}) ∧
    SubElements(e, {SocketFactory, Skeleton}) ∧
    SupportedFeatures(e, {Security}) ∧
    RequiredAttributes(e, ∅) ∧

    -- definition of sub-elements
        -- SocketFactory sub-element
        AllowedEAsForSubElement(e, SocketFactory, {SocketFactoryProvider}) ∧
        PropagateDeployment(e, SocketFactory) ∧
        -- Skeleton sub-element
        AllowedEAsForSubElement(e, Skeleton, {RMISkeleton}) ∧
        PropagateDeployment(e, Skeleton) ∧

    -- port bindings
        PortBinding(e, Skeleton, Factory, SocketFactory, Factory ) ∧
    -- port delegation
        PortDelegation(e, CallOut, Skeleton, CallOut) ∧
        PortDelegation(e, Mw, Skeleton, Mw) ∧

    -- signature propagation
    -- enforcing fixed signatures
    -- enforcing values of the realized features
    -- defining attributes referred in feature values
    -- feature delegation
        FeatureDelegation(e, Security, SocketFactory, Security)
)
```

Fig. 15 Example: image of SocketFactorySkeleton composite element architecture

```
∀e ∈ Elements : ElementHasEA(e, FileLogger) ⟹ (
    AllowedDeployment(e, {DockA}) ∧
    AvailablePorts(e, {CallIn, CallOut}) ∧
    SubElements(e, ∅) ∧
    SupportedFeatures(e, {Logging}) ∧
    RequiredAttributes(e, {FileName}) ∧

    -- signature propagation
        PropagateSignature(e, CallIn, CallOut) ∧

    -- enforcing fixed signatures

    -- enforcing values of the realized features
        FeatureHasValue(e, Logging, LoggingToFile) ∧

    -- defining attributes referred in feature values
        FeatureValueUsesAttribute(e, Logging, fileName, FileName)
)
```

Fig. 16 Example: image of FileLogger primitive element architecture

it is unique. This assumption is correct, as a connector theory describes a single connector.

As for the general distribution architecture constraints, the connector can feature only the roles defined by the communication style refined by the *RPC* distribution architecture (the *role* relation is constrained by *AvailableRoles*). Further, the cardinality of the roles is reflected (*roleInstance* is constrained by *RolesWithSingleCardinality* and *RolesWithMultipleCardinality*). Similarly, only the units defined by the RPC distribution architecture are allowed (*unit* is constrained by *AvailableUnits*), and their cardinality is reflected (*unit* is constrained by *UnitsWithSingleCardinality* and *UnitsWithMultipleCardinality*). Further, the connector may refer only to the features realized by *RPC* (*featureRequirements* is constrained by *SupportedFeatures*).

The next part of the template focuses on properties of the units. For each unit, the set of element architectures that can be employed for its refinement is explicitly prescribed (the *unit* and *elementArchitecture* relations are constrained by *AllowedEAsForUnit*). This ensures the refinement consistency of the units. Note that prescribing the element architectures for a unit determines its ports (as these element architectures have the same element type).

Further, the association of roles with units' ports is expressed (via constraining *role* and *unit* by *PortAssociatedWithRole*). The remote port bindings are reflected by forcing the signatures of the involved ports to be equal (*unit* and *ports* are constrained by *RemoteBinding*).

Finally, the feature delegation has to be captured (*unit* and *features* are constrained by *FeatureDelegation*).

5.4 Transforming requirements and deployment specifications

To illustrate this transformation of requirements and deployment specifications, we provide an example of the *CashDesk_to_Inventory* connector (Fig. 18); the template, as well as a

```
EmployedDA(RPC) ⟹ (
    AvailableRoles({Caller, Callee}) ∧
    RolesWithSingleCardinality({Callee}) ∧
    RolesWithMultipleCardinality({Caller}) ∧

    AvailableUnits({For_Caller, For_Callee}) ∧
    UnitsWithSingleCardinality({For_Callee}) ∧
    UnitsWithMultipleCardinality({For_Caller}) ∧

    SupportedFeatures( {Security}) ∧

    -- definition of units
        -- For_Caller unit
        AllowedEAsForUnit(For_Caller, {LoggedClientStub, RMIStub}) ∧
        -- For_Callee unit
        AllowedEAsForUnit(For_Callee, {SerializedServerSkeleton,
                                       SocketFactorySkeleton}) ∧

    -- association of units' ports with roles
        PortAssociatedWithRole(For_Caller, CallIn, Caller) ∧
        PortAssociatedWithRole(For_Callee, CallOut, Callee) ∧

    -- remote port bindings
        RemoteBinding(For_Callee, Mw, For_Caller, Mw) ∧

    -- feature delegation
        FeatureDelegation(Security, For_Callee, Security)
)
```

Fig. 17 Example: image of RPC distribution architecture

```
DefinedEndpoints({CashdeskCaller, InventoyCallee}) ∧
AllowedDAs( {RPC, LocalProcedureCall, ... }) ∧

-- definition of connector endpoints
    -- CashdeskCaller endpoint
    HasRole(CashdeskCaller, Caller) ∧
    HasSignature(CashdeskCaller, InventoryItf) ∧
    IsDeployedOn(CashdeskCaller, DockA) ∧
    -- InventoyCallee endpoint
    HasRole(InventoyCallee, Callee) ∧
    HasSignature(InventoyCallee, InventoryItf) ∧
    IsDeployedOn(InventoyCallee, DockB) ∧

-- definition of endpoints' features
    EndpointFeatureRequirements(CashdeskCaller, Logging, LoggingToFileFoo) ∧

-- definition of global connector features
    ConnectorFeatureRequirements( Security, SSL)
```

Fig. 18 Example: image of CashDesk_to_Inventory requirements specification

description of the transformation and template parameters, is given in [26].

5.4.1 Example: image of CashDesk_to_Inventory Requirements and Deployment

The image formulates constraints on the selection of predefined distribution and element architectures with respect to required features and deployment. Since communication style is the binding concept of requirements specification and predefined artifacts, the constraints are focused on the actual endpoints of the connector (i.e., actual roles' instances).

Therefore, it is first necessary to define the available endpoints of the connector (via constraining the *units* and *roleInstances* relations by *DefinedEndpoints*). Further, the set of

```
01   one sig Connector {
02     distributionArchitecture: one DistributionArchitecture,
03     units: UnitName one -> set Unit,
04     connectorFeatures: Feature set -> lone FeatureValue
05   }
06
07   abstract sig Element {
08     subElements: SubElementName lone -> lone SubElement,
09     ...
10   }
11   sig SubElement extends Element {}
```

Fig. 19 Example of a signature definition in Alloy

distribution architectures that can be employed for refinement of the connector is explicitly prescribed (*distributionArchitecture* is constrained by *AllowedDAs*).

As for each endpoint, its role is explicitly defined (*role* is constrained by *HasRole*), the signature of the associated component is assigned to the endpoint (*signature* is constrained by *HasSignature*), and its required deployment is enforced (via constraining *roleInstances* and *node* by *IsDeployedOn*).

Further, each feature requirement imposed directly on a particular endpoint is expressed (*roleInstances* and *features* are constrained by *EndpointFeatureRequirements*). Finally, the required values of the global connector features are enforced (via constraining *connectorFeatures* by *ConnectorFeatureRequirements*).

6 Arcas in alloy

In this section, after providing a very brief introduction to the Alloy [22–24] modeling language, we show how a connector theory can be expressed in Alloy. We also discuss the approaches for selecting an optimal CIC with the help of Alloy Analyzer.

6.1 Brief introduction to Alloy

This section gives a brief introduction to the Alloy modeling language—a formal modeling language based on a first-order predicate logic with operators from set theory (e.g., intersection, cartesian product), relational algebra (e.g., relational join, transitive closure), and basic arithmetics (e.g., integer operations, set cardinality) [22,23]. Further details of the current Alloy syntax can be found in [24].

The language is based on the notions of *signature* and *relation*. A signature is a set of abstract elements; relations are defined upon such sets. Alloy allows the constraint of relations by *facts* (first-order logic formulas). A fact can employ named predicates and function symbols. In general, an Alloy specification represents a first-order logic theory (*Alloy theory*[2]) determined by signature, relation, and fact definitions.

Alloy Analyzer, the associated solver, can either find a model of an Alloy theory, or check its models against a given property (expressed as a predicate). Alloy Analyzer converts the Alloy theory to a SAT formula; using an underlying general-purpose SAT solver, it solves the formula, and then maps the result to an Alloy theory model. Consequently, Alloy Analyzer requires the domains of signatures and relations to be explicitly bounded (due to the mapping to SAT).

In general, a definition of a signature `S` is interpreted in such a way that each of its fields `F` represents a relation between `S` and the signatures introduced by `F`. In a similar vein, *nesting* of signatures determines a subset relation on the signatures, while an *abstract* signature contains the elements of its nested signatures only.

In Fig. 19, the signature (`sig`) Connector is defined by listing relations `distributionArchitecture`, `units`, and `connectorFeatures`. Signature nesting is expressed by the `extends` keyword (line 11); `abstract` defines an abstract signature (line 7).

Obviously, the syntax complies with the object-oriented paradigm by defining signature as a structure constituting fields. Moreover, signature nesting resembles signature subtyping, whereas an abstract signature is akin to an abstract super-type.

A definition of a signature/relation may contain multiplicity constraints `one`, `lone`, `set`, `some`. Thus, the `Connector` set has to have exactly `one` element (i.e., a *singleton* signature). Similarly, `lone` denotes zero or one, while `some` at least one, and `set` zero or more elements. Thus, `distributionArchitecture` associates each element of `Connector` with exactly one element of `DistributionArchitecture`. Further, units is a relation between three sets: `Connector`, `UnitName` and

[2] In the Alloy documentation, Alloy theory is called Alloy model and a model of an Alloy theory is called Alloy model instance.

```
01   pred isSubElement[parent: Element, child: SubElement] {
02       child in univ.(parent.subElements)
03   }
04   fact ElementHierarchyIsTree {
05       -- There are no cycles among elements
06       no iden & ^{ parent: Element, child: SubElement |
07           isSubElement[parent, child] }
08       -- Every element which is not a unit has exactly one parent
09       all e: Element| e in SubElement <=>
10           one parent: Element | isSubElement[parent, e]
11   }
```

Fig. 20 Example of a predicate and fact definition in Alloy

`Unit`. For each `c` of Connector and each `un` of `UnitName` there is a subset *SU* (− > `set`) of Unit, so that for `u` ϵ *SU* the triple ⟨`c`, `un`, `u`⟩ is in units. Moreover, for each `c` and `u` there is exactly one `un` (`one`− >), so that ⟨`c`, `un`, `urlangle` is in units. The relation `connectorFeatures` is defined similarly.

A fact expresses a constraint over the sets and relations introduced by signature declarations. Each fact in an Alloy specification is an axiom of the theory determined by the specification.

In Fig. 20, the fact `ElementHierarchyIsTree` (lines 4–11) describes the properties of `subElements` using the predicate `isSubElement` (lines 1–3). In principle, it expresses that there are no cycles among the elements of `SubElement` with respect to the relation `subElements`, and that each element of `SubElement` has exactly one parent with respect to `subElements`.

The fact consists of two clauses (lines 6–7, 9–11) bound by a conjunction (expressed implicitly by a new line). To illustrate the basic Alloy constructs related to facts, consider the first clause (lines 6–7). Here, the identity relation (`iden`) is forced to have an empty (`no`) intersection (&) with the transitive closure (^) of the relation defined in the curly brackets. This relation contains all the pairs `parent` (of `Element`) and `child` (of `SubElement`), such that they satisfy the predicate `isSubElement`. Note that the operator imposes a constraint on its left operand by the predicate being its right operand.

As to other syntactic constructs, consider the `isSubElement` predicate (lines 1–3). The operator `in` stands for set/relation inclusion, while dot (.) denotes relational join; for example, `parent.subElements` results in a relation of the sets `SubElementName` and `Element`, such that for each of its tuples (`se`, `e`) the tuple (`parent`, `se`, `e`) is in `subElements`. Further, parent.subElements is prefixed by an outer join, the left operand of which is the entire domain (`univ`) of `SubElementName`. Consequently, the outer join yields a subset of `SubElement` (set being a special case of relation). Thus, the `isSubElement` predicate is satisfied if and only if `parent`, an arbitrary element of `SubElementName`, and child are in the `subElements` relation.

Reminiscent of object-oriented notation, relational join can be also expressed by [] (resembling indexing); thus, `univ.(parent.subElement)` can be rewritten as `parent.subElements[univ]`, or even `subElements[parent][univ]`. Note that [] also indicates the arguments of a predicate (e.g., `isSubElement`); the interpretation of [] depends on a particular context.

6.2 Connector theory in Alloy

In this section, we describe how a connector theory (Sect. 5) can be represented by means of Alloy. We focus on representing the meta-model of a connector theory (Fig. 10), describe a realization of its sets, and provide examples of how its constraints (Sect. 5) can be reflected in Alloy.

6.2.1 Representing the meta-model

Since the meta-model (Fig. 10) is based on sets and relations, it can be represented in Alloy in a straightforward way as illustrated below. For the purpose of the following examples we have modified the relations `features`, `attributes`, and `ports` so that their second and third fields are swapped (e.g., the domain of `features` becomes $Element \times Port \times Signature$ instead of $Element \times Signature \times Port$). The rationale for doing so is that this provides a more convenient Alloy syntax for expressing the constraints.

```
abstract sig DeploymentDock {}          abstract sig Port {}
abstract sig Role                       abstract sig LocalProvidedPort {}
abstract sig SubElementName {}          abstract sig LocalRequiredPort {}
abstract sig UnitName {}                abstract sig RemotePort {}
abstract sig Attribute {}               abstract sig Feature {}
abstract sig AttributeValue {}          abstract sig FeatureValue {}
abstract sig Signature {}               abstract sig ElementArchitecture{}
abstract sig DistributionArchitecture {}

abstract sig RoleInstance {
  signature: one Signature,
  role: one Role
}
abstract sig Element {
  elementArchitecture: one ElementArchitecture,
  ports: Port set -> lone Signature,
  subElements: SubElementName lone -> lone SubElement,
  dock: one DeploymentDock,
  features: Feature set -> lone FeatureValue,
  attributes: Attribute set -> lone AttributeValue
}
sig Unit extends Element {
    roleInstances: RoleInstance
}
sig SubElement extends Element {}
one sig Connector {
  distributionArchitecture: one DistributionArchitecture,
  units: UnitName lone -> set Unit,
  connectorFeatures: Feature set -> lone FeatureValue
}
```

The definition of the connector theory meta-model includes the integrity constraint ElementHierarchyIsTree articulated in Sect. 6.1. Note that the definition of the sets and relations also includes detailed cardinality constraints.

6.2.2 *Realizing meta-model sets*

Realization of the fixed sets by enumeration of all their elements is an inherent part of a connector theory. In Alloy, such an element is realized as a singleton set. As an example, the realization of DeploymentDock (Sect. 4.2.3) can take the following form:

```
one sig DockA extends DeploymentDock {}
one sig DockB extends DeploymentDock {}
```

The representation of FeatureValues is more complicated, since feature values may be hierarchical with parameters. In Alloy, we represent a parameter p by a relation p between two FeatureValues.

Moreover, in the context of an element architecture, the value of a parameter may be represented by an attribute and expected to be given later in the requirements specification (e.g., fileName in LoggingToFile in Sect. 4.1). This is expressed in Alloy by marking the associated signature by abstract (instead of one). This means that multiple concrete feature values are expected to be explicitly defined (each represented by a singleton subset), while the abstract signature defines their required structure. As an example, consider the LoggingToFile feature value defined in the FileLogger element architecture having the fileName parameter as its attribute.

```
abstract sig LoggingToFile extends FeatureValue {
    fileName: one FeatureValue      -- feature value parameter
}
```

A signature for a concrete value explicitly defines the value of the parameters. As an example, consider the LoggingToFile_InventoryLog (Sect. 4.2.2), assigning fileName to “inventory.log” value (represented by InventoryLog).

```
one sig InventoryLog extends FeatureValue {} -- stands for “inventory.log”
one sig LoggingToFile_InventoryLog extends LoggingToFile {} {
    fileName = InventoryLog  -- assignment of a particular parameter value
}
```

As for the alterable sets, the individual elements are created automatically by the Alloy Analyzer according to the corresponding constraints. Specifically for the `Signature` set, the created elements include concrete parameters of the signatures. The rationale is that while all the concrete values of the feature value parameters are explicitly given by the requirements specification, the concrete values of signature parameters depend on the actual composition and bindings of the elements, also created by the Alloy Analyzer. Technically, the Alloy representation of a parameterized signature is similar to the representation of a parameterized feature value—via a dedicated relation. The fact that signatures are to be created by the Alloy Analyzer is reflected by not marking the associated signature `abstract`. As an example, consider the `RMI` signature having the `javaSig` parameter (Sect. 4.1).

```
sig RMI extends Signature {
    javaSig: one Signature   -- signature parameter
}
```

6.2.3 Representing constraints (examples for element architecture)

First, the associated elements of the fixed sets have to be defined, including `Ports`, `SubElementNames`, `Features`, and `Attributes` sets; their elements are obtained by scanning the specification and the associated element type. For each element architecture, a singleton subset of the `ElementArchitecture` set is created. Thus, for the `FileLogger` element architecture (Sect. 4.1), the following will be created:

```
-- Ports
one sig CallIn extends LocalProvidedPort {}
one sig CallOut extends LocalRequiredPort {}
-- Features
one sig Logging extends Feature {}
abstract sig LoggingToFile extends FeatureValue {
       fileName: one AttributeValue
}
-- Attributes
one sig FileName extends Attribute {}
-- Element architecture
one sig FileLogger extends ElementArchitecture {}
```

The constraint itself is a direct representation of the clause of the connector theory illustrated in Fig. 16. In Alloy, a clause is represented by a fact. Compared to Fig. 16, the fact syntax in Alloy differs slightly in terms of logical operators, sets, and application of predicates (in addition, the conjunction after each predicate is in Alloy represented by a new line). Thus, for the `FileLogger` element architecture, the following Alloy fact will be created:

```
fact FileLogger {
    for all e: Element | ElementHasEA[e,FileLogger] => {
        AllowedDeployment[e, DockA]
        AvailablePorts[e, CallIn  + CallOut]
        SubElements[e, none]
        SupportedFeatures[e, Logging]
        RequiredAttributes[e, FileName]
        -- signature propagation
            PropagateSignature[e,CallIn,CallOut]
        -- enforcing fixed signatures
        -- enforcing values of the realized features
            FeatureHasValue[e, Logging, LoggingToFile]
        -- defining attributes referred in feature values
            FeatureValueUsesAttribute[e, Logging, fileName, FileName]
    }
}
```

As for predicates, to illustrate their representation consider the `AvailablePorts` predicate from Table 1. It is an example of a constraint on the second element of a ternary relation, where the third element is not relevant. Note that we have replaced the test for inclusion (from the associated formula in Table 1) in a relation with a relational join.

```
pred AvailablePorts[e: Element, availablePorts: set Port] {
    e.ports.univ = availablePorts
}
```

To illustrate the advantages of the Alloy language, below we show a representation of the `FeatureValueUsesAttribute` predicate from Table 1. Since a feature-value parameter is represented by a dedicated relation, we exploit the possibility of passing a relation as a parameter to an Alloy predicate. Here, relational join expresses the context of the constraint (e.g., that both `feature` value and `attribute` value are in the `features` and `attributes` relations with `e`).

```
pred FeatureValueUsesAttribute[e: Element, feature: Feature,
    parameter: FeatureValue -> AttributeValue, attr: Attribute] {
        e.features[feature].parameter = e.attributes[attr]
}
```

Compared to the corresponding formula from Table 1, it is obvious that the Alloy representation is much more concise and comprehensive, as it resembles an expression in a regular object-oriented programming language.

In a similar vein, `PortBinding` below is more concise than the corresponding predicate from Table 1. Here, the relational join allows expressing the context of the constraint transitively (combining the relations `subElements` and `ports`).

```
pred PortBinding[e: Element,
    subelement1: SubElementName, port1: LocalRequiredPort,
    subelement2: SubElementName, port2: LocalProvidedPort] {
        e.subElements[subelement1].ports[port1]
        = e.subElements[subelement2].ports[port2]
}
```

6.3 Selecting an optimal CIC

The set of models of a connector theory can be of a large cardinality; nevertheless, just a single model is needed. Thus, it is necessary to make a choice and select the "best" one (the best CIC). A practical necessity is to automate the selection process.

There are different criteria for judging what the "best" CIC is, ranging from memory consumption and CPU utilization, latency and throughput, to robustness, reliability, and stability, represented by means of valuation of the elements involved in CIC. Naturally, it is necessary to find rules for composability of the criteria. This leads to an optimization problem where the task is to find an optimal CIC given a valuation of its elements (their element architectures in particular) by applying the rules. A simple example of such a valuation is a manual assignment of a fixed cost to each element architecture where the valuation of CIC is the sum of element-architecture costs for all the elements in it. The CIC with the lowest cost is pronounced the "best".

The Alloy language itself does not provide any explicit support for solving optimization problems. In principle, there are three possible approaches to employing the Alloy framework for finding an optimal CIC: (i) to enumerate all the models of a given connector theory and select an optimal one by applying valuation criteria outside the resolution process (an enumeration of all models is a standard feature of the Alloy Analyzer); (ii) to encode the optimization problem into a "standard" Alloy theory; and (iii) to extend the Alloy language and Alloy Analyzer accordingly.

We do not consider other approaches, such as extending the Alloy Analyzer without changing the Alloy language (i.e., instrumenting the Analyzer with custom heuristics), or moving from Alloy to another constraint-solving technique. Preferring relative simplicity and efficiency, we have used approach (i) for the experiments and evaluation.

6.3.1 Enumerating all connector theory models

A standard feature of the Alloy Analyzer is the enumeration of all models of a given Alloy theory by incremental execution of its underlying SAT solver. This feature can be effectively exploited for finding an optimal CIC. Here, the Alloy Analyzer finds all the models of a given connector theory and a valuation of the models is provided by a dedicated external tool (i.e., a total order upon the models is created); finally, an optimal CIC is determined based on the partial order.

By delegating the optimization to a dedicated external tool, this method does not require expressing optimization criteria in a connector theory. Consequently, the model valuation strategy is not limited by the expressive power of the Alloy language, but it is entirely determined by the options the external tool provides.

Even though it is necessary to explore all the models, this method is still tangible, since each of the models is evaluated separately.

6.3.2 Encoding optimization problems in Alloy

Modeling optimization problems in Alloy is based on expressing a partial order over the set of CICs where the "best" CIC is the least element. Since the order of a particular CIC is typically determined by the employed element architectures, it is necessary to define a global partial order of all the available element architectures, as well as a way of inferring the CIC partial order from this global order.

Defining such a way of inferring is a challenge. Since the use of natural numbers provided by Alloy is not feasible due to the increased complexity, the standard Alloy relations have to be used. Here, it is necessary to assess the order of composite element architectures based on the order of their sub-element architectures, which is also a challenge. Another option is also to employ the state-of-the-art alternatives for solving Alloy theories [13,14] that offer better support for arithmetic and unbounded integer types.

6.3.3 Extending the Alloy framework

The Alloy language could be extended to allow for specifying optimization criteria. The Alloy Analyzer would have to translate such optimization criteria into a SAT formula. Specifically, such an extension can be achieved by employing pseudo-Boolean (PB) formulas instead of SAT formulas. Here, an assumption is that the Alloy Analyzer supports a PB solver such as MiniSAT+ [15] (this appears to be realistic in the future, since the MiniSAT solver is already supported by Alloy Analyzer). Overall, such an extension of the Alloy framework requires a major modification of its current implementation.

7 Related work

In general, the related work spans three areas: (i) composing software with the help of constraint solving, (ii) Alloy-based resolution and verification of component architectures, and (iii) automated connector synthesis.

(i) Composing software with the help of constraint solving. Constraint solving techniques have been already used for a variety of tasks in software composition ranging from automated dependency management [28] to verification of composition in product lines [37]. In [28], the idea is to employ a SAT solver to resolve dependencies; this approach was used in several contemporary software tools such as OSGi implementation Equinox p2 and Maven.[3]
The approach presented in [37] is based on formalizing the notion of a feature model by introducing a specific feature algebra. Having such formalization available, a SAT solver is employed for verification of safe feature-model composition in a product line.
Compared to ARCAS, both methods leverage on simple propositional formulas for expressing the given problem. Such formulas merely express variability and transitivity, but do not reflect more complex properties of the software parts to be composed (e.g., interface bindings, runtime environment requirements, and features).

(ii) Alloy-based resolution and verification of component architectures. Alloy has been already extensively used in the domain of CBSE for the purpose of both property checking and model finding. In [16], the authors examine the feasibility of using architectural constraints as the basis for specification, design, and implementation of self-organizing architectures in Darwin. In this context, Alloy serves as a tool for an automated resolution of the self-organizing reconfigurations from an inconsistent to a consistent state in terms of a particular architectural style.
In a similar way, [17] employs Alloy for specification of the possible architecture reconfiguration actions in the context of a generic component model based on OSGi. Alloy is employed in two ways: (a) architecture change verification and (b) architectural change planning. The former focuses on soundness of the reconfiguration actions and preservation of the properties specific to a particular architectural style. The latter comprises finding a fitting sequence of reconfiguration actions from the current consistent state to a given consistent state while preserving a particular architectural style in all intermediate states.

[3] http://www.eclipse.org/equinox/p2/ and http://maven.apache.org/.

A methodology for verifying soundness of self-configuration scenarios via their specification in Alloy is presented in [39]. The underlying formal method, FracToy, formalizes a concrete self-configurable system, as well as the corresponding self-organizing actions. The verification targets both static and dynamic properties. Compared to our approach, models of the Alloy theory only prove the consistency/soundness of the theory and are not used for any other purpose.
In [27], Alloy is employed for the formal specification of various architectural styles. The main goal is to check the important properties of an architectural style such as consistency, satisfaction of a predicate over an architectural style instance, composability, and refinement of architectural styles. The Alloy formalization of an architectural style is obtained programmatically from its Acme specification (which servers a similar purpose as our CDL specification).
Apart from reconfigurations, in [25], a formal specification of valid component compositions in the COM component model is analyzed, while in [31] a fully fledged Alloy formalization of the Fractal component model is presented.

(iii) Automated connector synthesis. The ARCAS method stems from our previous research [2,4,6,7]. In [2,7], a connector model designed for automated connector synthesis based on a high-level specification is presented. Despite leveraging on similar concepts to those presented in Sect. 2, it does not explicitly capture NFPs. The key difference lies in the way a connector configuration is selected—it is performed via term matching in Prolog. Breaking the separation of abstraction levels, the method imposes the inclusion of Prolog terms directly in the requirements, deployment, and artifact specifications.
An extensive effort has been put into research of automated synthesis of connectors ensuring application-layer and middleware-layer interoperability [19]. For brevity, we call the former API mediation and the latter middleware bridging. While we have focused on cases where a component/service being deployed is to be connected to another one (potentially already deployed and running), Issarny et al. [19] exploit a slightly different scenario—connecting solely the already deployed and running components/services. The connected components may require both API mediation and middleware bridging. Thus, letting NFPs aside, in [19], the input is an API specification and a determination of a particular middleware for each of the connected components; the goal is to synthesize (in an automated way) a connector implementation, that does the mediation and/or bridging. For comparison, in ARCAS the input is a deployment and requirements specification; the goal is to find a fitting connector implementation employing a suitable

middleware/middleware bridging (a potential API mediation is expected to be done by a separate component).

In [20], an automated method for generating connectors is proposed to be employed for synthesis of so-called emergent connectors. Here, "emergent" refers to synthesis at runtime without any preceding design. Specification of such emergent connectors is to be programmatically derived from the specification of components to be connected [19]. In comparison, ARCAS allows for emergent connector synthesis solely in terms of synthesis at deployment time and runtime and, unlike [19], it has to be provided with (at least) a high-level (requirements and deployment) specification of the connector being synthesized.

In [35], the concept of a connector is introduced for CCM. The goal is to benefit from a light communication middleware—lighter than CORBA—in CCM-based applications by introducing connectors (strictly separating the component communication from business logic). The connector-generation process employs an extended IDL for defining connector templates. These are, however, not composable. Similar to ARCAS, connector specification is accompanied by a description of connector-specific features in an extended OMG D&C specification. Connectors are generated by manually selecting a particular template parameterized by the actual interfaces; the connector templates have also to be created manually.

In [34], targeting model-based synthesis of component-based applications from UML-MARTE models, the concept of compositional connectors is employed as well. Here, a connector architecture has to be explicitly captured by a UML-MARTE model (created manually), while in ARCAS the connector's architecture is determined automatically using constraint solving. Nevertheless, the paper mentions the idea of a semi-automatic, deployment-time selection of an implementation for specific parts of the connector according to platform-related meta-information.

8 Discussion and evaluation

In this section, we discuss the important decisions and trade-offs we had to make and mention open issues we still face. In particular, these include (i) interpretation of a connector theory model, (ii) focus of a connector theory, (iii) addressing NFPs, (iv) moving ARCAS to other domains, (v) choice of Alloy, (vi) experience and case study, and (vii) issues to be addressed.

(i) Model interpretation. The ARCAS method exploits the Alloy Analyzer's capability of finding a model of a given Alloy theory. In contrast to a typical usage of this feature—interpreting the existence of a model as consistency/soundness of the theory and using such a model as feedback while developing a theory, e.g., [1,39]—the ARCAS method directly interprets/employs a model as the desired CIC.

(ii) Focus of connector theory. A connector theory in ARCAS differs from similar formal models [31] in its focus. While in [31] focuses on the meta-model level, ARCAS concentrates on the model level by specifying semantics of a particular connector via concrete instances of meta-model entities (e.g., `FileLogger` as an instance of element architecture). This is facilitated by the automated transformation of the connector specification into a connector theory, whereas formal models at the meta-model level are typically created manually. The ARCAS method thus provides the option of reasoning about a particular connector instead of reasoning solely about properties common to all possible connectors.

(iii) Addressing NFPs. The ARCAS method makes it possible to address NFPs in the synthesized connector. This is in contrast to some of the state-of-the-art methods for automated synthesis of middleware-based connectors [19]. Since connector theory also comprises element composability with respect to NFPs, it allows programmatic synthesis of a connector complying with the NFP requirements. Nevertheless, since the Alloy Analyzer lacks a specialized support for arithmetic operators, it is feasible to address only qualitative NFPs (i.e., those expressible by enumeration), whereas efficient addressing of quantitative NFPs (such as latency) would be hard to achieve. However, there are constraint-solving techniques providing advanced support for arithmetic operations and working with inequalities [12], which might be sufficient for addressing quantitative NFPs. In this respect, a challenging issue is to integrate these constraint-solving techniques with the Alloy framework, or adapt the ARCAS method into another framework supporting such techniques.

(iv) Moving ARCAS to other domains. Although the ARCAS method has been presented as having its principal application in automated resolution of middleware-based connectors, we argue that it is a general method for automated composition of hierarchical architectures based on constraint solving, applicable in other domains. From this perspective, its application to connectors is merely a specific case. In general, ARCAS is applicable in a domain with the following key characteristics: (i) it employs hierarchical component-based architectures, (ii) if NFPs are required, they can be articulated in a composable way, (iii) the related specifications can be transformed to a theory (i.e., a constraint specification) programmatically, (iv) the theory can be articulated in such a way that automated model-finding is possible, and (v) criteria for selecting an optimal model of a theory can be formed.

Specifically, the idea of ARCAS—automated composition of hierarchical elements based on constraint solving—universally applies to automated composition of architectural patterns based on pipe-and-filter (including their nesting). A typical example illustrating such a potentially nested pattern is a media player. Such a player employs a number of codecs (audio and video), filters, muxers, and demuxers, which have to be correctly organized in an architecture in order to process the content from an input stored in a file or available on-line. From the ARCAS perspective, the whole architecture of a media player can be likened to a connector, and each of the codecs, filters, muxers, and demuxers can be likened to a connector element. The key property of ARCAS, applicable in this scenario, is the automated assembly of the elements while ensuring their compatibility and meeting requirements on NFPs. Thus, it is possible, for example, to handle the tradeoff between computational complexity and video image quality.

Another domain where ARCAS can be applied is the domain of embedded real-time systems. Here, ARCAS can provide an automated selection of hardware sensors, corresponding device drivers, and proper API of the operating system. In analogy with a connector, each sensor, device driver, and particular API variant can be likened to an element. The option of specifying NFPs can be employed, e.g., for determining the required sampling rate of a sensor.

(v) Choice of Alloy. The choice of Alloy is motivated by its expressive power of relational first-order predicate logic and its convenient, object-like syntax (Sect. 6.1). An advantage is the integration of the Alloy Analyzer in Java. On the other hand, a limitation is the requirement on bounded domains of all sets and relations in an Alloy specification (stemming from the underlying usage of SAT, Sect. 6.1) which in case of ARCAS implies the need of assessing the number of elements in a connector prior to the actual connector theory resolution. Nevertheless, this assessment can be derived from the available artifact specifications with the help of incremental execution of the Alloy Analyzer.

(vi) Experience and case study. As a proof-of-the-concept, we have developed an EMF[4]-based demonstrator[5] involving the transformations of specifications and integrating Alloy Analyzer. In addition, we have developed an experimental database of connector artifacts (including both the specifications and their Alloy images). The examples from this text are simplified versions of the artifacts in this database. We have employed this database in a case study involving a non-trivial part of a real-life component-based application based on the procedure-call communication style [18]. Various client–server connection scenarios differing in NFPs and deployment were considered. This helped demonstrate the soundness and feasibility of the Alloy-based CIC resolution on a real-life example. In addition to procedure call, we have also successfully modeled and evaluated all the other common communication styles relevant to middleware-based connectors—asynchronous messaging, blackboard, and streaming. These (including the procedure call) cover all the connector types that the taxonomy [29] categorizes as "communication".

We have also performed several benchmarks in order to assess the performance scaling factors of ARCAS. For this purpose, the actual database of connector artifacts was generated in an automated way by cloning and introducing new variants in the original database. This technique closely mimics a general case, since the performance of Alloy Analyzer mainly depends on the cardinality of the sets and relations rather than on the complexity and variability of the constraints. Based on our measurements, even though the computational complexity is NP-complete in principle (Alloy Analyzer employs a SAT solver), ARCAS is feasible up to hundreds of element architectures and tens of distribution architectures. Specifically, for 100 element architectures and 10 distribution architectures the execution times are in the order of 5 s (Alloy set up for MiniSAT and 512MB on Intel i5 2.6 GHz); for 200 element architectures and 20 distribution architectures the execution time is around 14 s. The performance can be further improved by representing connector theory directly in Kodkod [40] (the underlying relational solver) instead of Alloy. Note that the numbers above (100/200) pertain just to the element architectures and distribution architectures that are applicable for a particular connector (these architectures were selected from a much larger database).

(vii) Issues to be addressed. The ARCAS meta-model does not reflect (a) cardinality of ports and sub-elements (e.g., important when multiple client stubs have to be served by a single server skeleton, not forcing serialization of requests), and (b) composite port signatures (a typical phenomenon when a server unit supports a number of middleware protocols simultaneously). In this paper, these concepts were left out for simplicity, but the ARCAS method can be enhanced to support both (a) and (b). Addressing (a) comprises an extension to the element type and element architecture abstract

[4] http://www.eclipse.org/modeling/emf/.

[5] http://d3s.mff.cuni.cz/projects/components_and_services/arcas/.

syntax, as well as straightforward modifications of the transformation of element architectures and distribution architectures. Addressing (b) involves modifications of the connector theory and associated transformations. As an aside, both (a) and (b) have been already experimentally proved feasible.

Since the behavior of middleware-based connectors is rather simple and driven by the communication style, the responsibility of the behavioral compliance between element types and element architectures is upon the elements' designer.

9 Conclusion and future work

In this paper, we presented a method for automated resolution of connector architectures based on constraint solving—the ARCAS method. An important benefit of ARCAS is the ability to address the required NFPs, which, as well as transparent distribution, is the major concern of middleware-based connectors. In ARCAS, we assume middleware connectors are based on hierarchical elements, similar to hierarchical components. This allows definition of the individual parts of a connector in advance, and thus facilitates reuse. The key idea of ARCAS is to resolve a description of a particular connector instance (CIC) as a model of a theory based on a first-order logic and relational calculus—a connector theory. We have defined automated transformations, which convert the predefined connector artifact, requirements, and deployment specifications to such a connector theory. Overall, ARCAS can be employed whenever the requirements or deployment changes (even at runtime). Moreover, we have articulated the characteristics another domain would have to satisfy to make ARCAS applicable.

As a proof-of-the-concept, we described the representation of a connector theory in the Alloy modeling language. Moreover, using the Alloy representation, we have shown the feasibility of ARCAS on a real-life example. We have also developed a demonstrator involving the transformations of specifications and integrating Alloy Analyzer.

In our future work, we intend to focus on providing support for modeling optimization problems in Alloy, as well as on introducing support for quantitative NFPs (either by extending the Alloy framework or employing another constraint solver framework). Finally, we plan to explore the possibility of integrating state-of-the art methods for middleware and application interoperability [8,19,21] to achieve a resolution-based synthesis of emergent connectors.

Acknowledgments This work was partially supported by the EU project ASCENS 257414 and by the Grant Agency of the Czech Republic, project GACR P202/10/J042. We are also very grateful to the anonymous reviewers for their valuable comments and suggestions and to Hugo Gibson for proofreading the text.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Softw. Syst. Model **9**(1), 69–86 (2008)
2. Bures, T.: Generating Connectors for Homogeneous and Heterogeneous Deployment, PhD dissertation. Charles University in Prague, Dept. of Distributed and Dependable Systems (2006)
3. Bulej, L., Bures, T.: Addressing heterogeneity in OMG D&C-based deployment, Tech. Report No. 2004/7, Dep. of SW Engineering, Charles University, Prague. http://d3s.mff.cuni.cz/publications/ (2004)
4. Bulej, L., Bureš, T.: Deploying Heterogeneous Applications using OMG D&C and Software Connectors, Tech. Report No. 2005/10, Dep. of SW Engineering, Charles University, Prague. http://d3s.mff.cuni.cz/publications/, November (2005)
5. Benmokhtar, S., Georgantas, N., Issarny, V.: COCOA: COnversation-based service COmposition in pervAsive computing environments with QoS support. J. Syst. Softw. **80**, 1941–1955 (2007)
6. Bures, T., Hnetynka, P., Plasil, F.: SOFA 2: Balancing Advanced Features in a Hierarchical Component Model, Proceedings. of 4th International Conference on Software Engineering Research, Management and Applications (SERA '06). IEEE Computer Society, Washington, DC (2006)
7. Bures, T., Plasil, F.: Communication style driven connector configurations. In: Software Engineering Research and Applications. Springer, Berlin (2004)
8. Blair, G., Paolucci, M., Grace, P., Georgantas, N.: Interoperability in complex distributed systems. In: Bernardo, M., Issarny, V. (eds.) SFM 2011, LNCS, vol. 6659. Springer, Heidelberg (2011)
9. Bliudze, S., Sifakis, J.: The algebra of connectors: structuring interaction in BIP. In: Proceedings of EMSOFT '07, pp. 11–20. ACM, New York (2007)
10. Cubo, J., Canal, C., Pimentel, E.: Context-aware composition and adaptation based on model transformation. J. Universal Comput. Sci. **17**, 777–806 (2011)
11. Crnkovic, I., Larsson, M.: Building Reliable Component-Based Software Systems. Artech House, Norwood (2002)
12. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer, Berlin (2008)
13. El Ghazi, A.A., Taghdiri, M.: Analyzing alloy constraints using an SMT solver: a case study. In: 5th International Workshop on Automated Formal Methods (AFM). Edinburgh (2010)
14. El Ghazi, A., Taghdiri, M.: Relational Reasoning via SMT Solving, FM 2011: Formal Methods, pp. 133–148. Springer, Berlin (2011)
15. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into SAT. J. Satisfiability Boolean Model. Comput. **2**, 1–26 (2006)
16. Georgiadis, I., Magee, J., Kramer, J.: Self-organising software architectures for distributed systems. In: Proceedings of the First Workshop on Self-Healing Systems, New York (2002)
17. Hansen, K.M., Ingstrup, M.: Modeling and analyzing architectural change with alloy. In: Proceedings of the 2010 ACM Symposium on Applied Computing-SAC '10, p. 2257 (2010)
18. Herold, S., Klus, H., Welsch, Y., et al.: CoCoME-the common component modeling example. The Common Component Modeling Example, pp. 16–53 (2008)
19. Issarny, V., Bennaceur, A., Bromberg, Y.D.: Middleware-layer Connector Synthesis: Beyond State of the Art in Middleware Interoperability. In: Bernardo, M., Issarny, V. (eds.) Formal Methods

for Eternal Networked Software Systems, pp. 217–255. Springer, Berlin (2011)

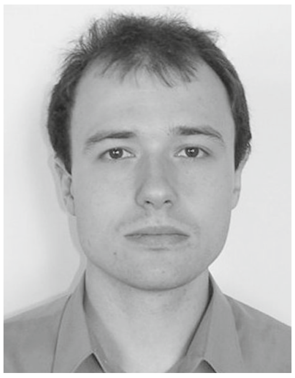
20. Issarny, V., Steffen, B., Jonsson, B., Blair, G., Grace, P., Kwiatkowska, M., Calinescu, R, Inverardi, P., Tivoli, M., Bertolino, A., Sabetta, A.: CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In: 2009 14th IEEE International Conference on Engineering of Complex Computer Systems, pp. 154–161 (2009)
21. Inverardi, P., Spalazzese, R., Tivoli, M.: Application-layer connector synthesis, Formal Methods for Eternal Networked Software Systems, pp. 148–190. Springer, Berlin (2011)
22. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**, 256–290 (2002)
23. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
24. Jackson, D.: Alloy Language Reference. http://alloy.mit.edu (2011)
25. Jackson, D., Sullivan, K.: COM revisited: tool-assisted modelling of an architectural framework. ACM SIGSOFT Softw. Eng. Notes **25**, 149–158 (2000)
26. Keznikl, J., Bureš, T., Plášil, F., Hnětynka, P.: Automated Resolution of Connector Architectures Using Constraint Solving (ARCAS method)", Tech. Report No. D3S-TR-2012-03, Dep. of Distributed and Dependable Systems, Charles University, http://d3s.mff.cuni.cz/publications/ (2012)
27. Kim, J.S., Garlan, D.: Analyzing architectural styles with alloy. In: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis-ROSATEA '06 (2006)
28. Le Berre, D., Parrain, A.: On SAT Technologies for Dependency Management and Beyond. In: Proceedings of 12th International Software Product Line (SPLC 2008) vol. 2, pp. 197–200 (2008)
29. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: Proceedings of the 22nd International Conference on Software Engineering. ACM (2000)
30. Malohlava, M., Plášil, F., Bureš, T., Hnetynka, P.: Interoperable DSL Families for Code Generation, Tech. Report No. D3S-TR-2011-04, Dep. of Distributed and Dependable Systems, Charles University, Prague. http://d3s.mff.cuni.cz/publications/, April (2011)
31. Merle, P., Stefani, J.B.: A formal specification of the Fractal component model in Alloy, Research Report RR-6721, INRIA. http://hal.inria.fr/inria-00338987/en/ (2008)
32. Nakazawa, J., Tokuda, H., Edwards, W.K., Ramachandran, U.: A Bridging Framework for Universal Interoperability in Pervasive Systems. In: 26th IEEE International Conference on Distributed, Computing Systems (ICDCS'06), p. 3 (2006)
33. Object Management Group: Deployment and Configuration of Component-based Distributed Applications Specification. http://www.omg.org/cgi-bin/doc?formal/06-04-02.pdf, Feb (2004)
34. Radermacher, A., Cuccuru, A., Gerard, S., Terrier, F.: Generating Execution Infrastructures for Component-Oriented Specifications with a Model Driven Toolchain: A Case Study for MARTE's GCM and Real-Time Annotations, pp. 127–136. ACM, New York Proceedings of the eighth international conference on Generative programming and component engineering (2009)
35. Robert, S., Radermacher, A., Seignole, V., Gérard, S., Watine, V., Terrier, F.: Enhancing interaction support in the corba component model, From Specification to Embedded Systems Application, pp. 137–146 (2005)
36. Spalazzese, R., Inverardi, P.: Mediating Connector Patterns for Components Interoperability. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 335–343. Springer, Heidelberg (2010)
37. Thaker, S., Batory, D., Kitchin, D., Cook, W.: Safe composition of product lines. In: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, pp. 95–104. ACM, New York (2007)
38. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. Wiley, Hoboken (2010)
39. Tiberghien, A., Merle, P., Seinturier, L.: Specifying Self-configurable component-based systems with FracToy. Abstract State Mach. Alloy B Z **5977**, 91–104 (2010)
40. Torlak, E.: A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications. Ph.D. Thesis, MIT, February (2009)

Author Biographies

Jaroslav Keznikl graduated from Charles University in 2010 with a major in the computer science. Currently, he is a PhD candidate at Department of Distributed and Dependable Systems, Charles University in Prague. He participates in a number of research projects including SOFA 2 and EU FP7 ASCENS. His research interests include software architectures, formal modeling of software systems, model-driven development, and component-based development.

Tomáš Bureš is an assistant professor at the Department of Distributed and Dependable Systems (D3S) of Charles University Prague. He received his PhD degree in 2006 also from Charles University. In the meantime, he held 1 year post-doctoral researcher position at MSlardalen University, Sweden. He specializes in component-based development, especially software connectors and generative programming. He has participated in a number of projects including SOFA and SOFA 2, Q-ImPrESS, ITEA/EUREKA projects OSMOSE, OSIRIS, and EU FP7 ASCENS and in a research project with France Telecom.

František Plášil is professor of software engineering with Department of Distributed and Dependable Systems (D3S). In his research, he focuses on component-based and service-oriented programming, and also on behavior specification and verification for software components. He led D3S teams in several research projects such as ITEA OSMOSE, ITEA OSIRIS, EU FP7 Q-ImPrESS and ASCENS. He co-authored over 100 refereed articles in international journals and proceedings of international conferences, and also served on the program committees of numerous international conferences, and editorial boards of several international journals. Professor Plasil has also taught at Universiy of Denver, USA; Wayne State University, USA; University of New Hampshire, USA; and University of Linz, Austria.



Petr Hnětynka is an assistant professor at the Department of Distributed and Dependable Systems (D3S) of Charles University Prague. He received his PhD degree in 2005 also from Charles University. In the meantime, he held 1 year postdoctoral researcher position at University College Dublin. He specializes in component-based development, especially a design of component models and model-driven development. He has participated in a number of projects including SOFA and SOFA 2, Q-Impress, ITEA/EUREKA projects OSMOSE, OSIRIS, and EU FP7 ASCENS.

3.5 Towards Verification of Ensemble-Based Component Systems

Jiří Barnat,
Nikola Beneš,
Tomáš Bureš,
Ivana Černá,
Jaroslav Keznikl,
František Plášil

In proceedings of the **10th International Symposium on Formal Aspects of Component Software (FACS '13)**.

In *Lecture Notes in Computer Science,*
Volume 8348,
to be published by Springer,
October 2013.

Summary of the Paper

This paper, which was published as [BBB+13], focuses on the formal aspects of the DEECo component model (presented in Sections 3.1 and 3.2) and the resulting challenges for verification of DEECo-based applications so as to respond the need of dependability in RDS (i.e., C4 in Section 1.3).

To this end, while pursuing the research goal **G3**, the key contribution of this paper lies in discussing the benefits and limitations of model checking in the context of DEECo-based systems, specifically focusing on the challenges stemming from architecture dynamism and parallel, soft-real-time execution of component processes and ensemble knowledge exchange. (Note that DEECo is presented under the umbrella of Ensemble-Based Component Systems, similar to Sections 3.2 and 3.3). In particular, the paper presents a formalization of the general operational semantics of DEECo (outlined in Section 3.2) and examines it w.r.t. the properties analyzable via model checking. This is done by mapping the semantics onto DCCL [BBCP13], which is a semantic model embodying abstractions suitable for LTL model checking [VW86] of ensemble-based systems via the model-checker DiVinE [BBH+13]. However, DCCL introduces additional simplification and abstraction so as to make the model checking feasible.

Based on previous work [AABG+13], the paper formalizes the general operational semantics of DEECo in a way that faithfully captures the operation of a DEECo-based application and its execution platform in a real environment. Specifically, it accounts for (a) fully asynchronous, distributed, and decentralized operation of components and ensembles, (b) real-time scheduling, and (c) network-specific issues such as communication delays and losses. The general semantics is tailored for further specialization, i.e., it is general enough that other DEECo semantics aimed at verification (e.g., based on DCCL) or stemming from different implementations (e.g., by employing different communication middleware) can be readily presented as specializations of the general semantics. Technically, the general semantics of DEECo is formalized using a finite-state non-deterministic automaton, in which the states capture component knowledge and the transitions correspond to execution of component processes or ensemble knowledge exchange. Since DEECo-based applications are in general soft-real-time systems, the traces generated by the automaton are further restricted so as to reflect only the realistic behaviors w.r.t. real-time periodic scheduling (e.g., each process has to start and end within each period). In terms of model checking, these restrictions impose specific "fairness" constraints [CGP99] on the automaton traces.

The paper then demonstrates that the DCCL model-checking semantics is a specialization of the general DEECo semantics – meaning that DCCL generates only a subset of execution traces permitted by the general semantics and that a violation of a property under DCCL implies violation of an equivalent property under the general semantics. Further, the paper identifies and discusses the properties that are legitimate candidates for verification under the general DEECo semantics and can be also verified under the model-checking semantics. Note that the properties are discussed on a rather concrete, DEECo-specific level (e.g., resilience w.r.t. knowledge inconsistency caused by knowledge exchange/process parallelism). On the other hand, the paper also identifies and discusses several limitations of the model-checking semantics both in terms of verifiable properties, as it introduces additional abstraction, and performance, as the highly concurrent and dynamic DEECo-based systems generate a very large state space.

As a bottom line, the paper shows that even after introducing relatively significant simplifications to the operational semantics (e.g., unlimited time for knowledge exchange), it is still possible to maintain a rich set of verifiable properties. However, even with the simplifications, the high level of dynamism and parallelism still prevents scaling the model checking to bigger scenarios. Consequently, this makes verification of the properties that were abstracted away due to the simplifications a bigger challenge still.

To provide a concrete illustration and evaluation of these conclusions, the paper presents a realistic case study stemming from the cooperative vehicle navigation scenario featured by the ASCENS project [SRA+11].

This paper also forms the basis for the current work on an operational semantics following a fully decentralized computational model [BGH+14a].

Comments on Authorship

While the core ideas of this paper stem from the collaboration with the other authors, primarily my supervisor, my personal contribution lies in elaborating the main idea, which includes resolving the relationship between DEECo and DCCL and identification of the verifiable properties. I was also responsible for detailing the DEECo computational model and its formalization. Further, I conducted the case study. Finally, again under helpful guidance and supervision of the other authors, I authored a majority of the text.

Towards Verification of Ensemble-Based Component Systems

Jiří Barnat[1], Nikola Beneš[1(✉)], Tomáš Bureš[2], Ivana Černá[1], Jaroslav Keznikl[2], and František Plášil[2]

[1] Faculty of Informatics, Masaryk University, Brno, Czech Republic
{xbarnat,xbenes3,cerna}@fi.muni.cz
[2] Faculty of Mathematics and Physics, Charles University in Prague, Praha, Czech Republic
{bures,keznikl,plasil}@d3s.mff.cuni.cz

Abstract. The relatively new domain of Ensemble-Based Component Systems (EBCS) brings a number of important verification challenges that stem mainly from the dynamism of EBCS. In this paper, we elaborate on our previous work on EBCS verification. In particular, we focus on verification of applications based on the DEECo component model – a representative of EBCS – and evaluate it on a real-life case study. Since our verification technique employs a specialized DEECo semantics to make the verification problem tractable, our goal is to investigate the practical relevance of the properties that can be addressed by the verification. Specifically, we compare the specialized semantics with the realistic general semantics of DEECo to identify verification properties that are preserved by the specialized semantics. We further investigate the tractability of verification of these properties on a real-life case study from the domain of electrical vehicle navigation – one of the key case studies of the EU FP7 project ASCENS.

Keywords: Component-based systems · Component ensembles · Formal verification

1 Introduction

Ensemble-Based Component Systems (EBCS) is a new class of component-based systems, characterized by the fact that the "traditional" component architecture based on explicit bindings is replaced by a composition of components into so-called *ensembles* [6,8]. An ensemble is a first-class concept that addresses the dynamism in software architecture by declaratively capturing the component composition and the corresponding interaction. In particular, this is done by identifying the components to be composed implicitly via a predicate over component states, so that each group of components for which the predicate holds forms an ensemble, and by describing the interaction among the components via a mapping relation among the states of these components. Furthermore, to compensate for the lack of the global system view, the components in EBCS are autonomic entities building on agent-oriented concepts [12] and featuring execution model based on feedback loops (e.g., soft real-time control systems [9]) in order to achieve (self-) adaptive and resilient operation. As an aside, following the agent-oriented paradigm, in EBCS the state of a component is called the component's knowledge. EBCS are thus very appropriate for design and development of highly dynamic autonomous systems that heavily interact with the physical environment – in literature typically termed Cyber-Physical Systems (CPS) [10].

In our previous work, we have introduced a representative of EBCS – the DEECo component model [4,7] (Dependable Emergent Ensembles of Components). In addition to reification of EBCS concepts and language mapping to Java, DEECo comes also with a well-defined semantics [1], which reflects the need for distributed and fully decentralized operation while specifically dealing with components, ensembles, and knowledge. DEECo's semantics is intentionally very general to allow for a number of compliant realizations (i.e., specializations generating strict subsets of traces allowed by the general DEECo semantics) by means of different communication middleware.

The generality of DEECo's semantics however brings about the problems of generating an extensive state space, which is intractable for verification of DEECo-based systems. To alleviate this restriction, we have come up with the Dynamically Communicating Components Language (DCCL) [3] – a specialization of the DEECo's semantics, which by sacrificing some variability in the general DEECo's semantics significantly reduces the state space and thus makes model-checking of DEECo-based applications tractable. Systems described in DCCL can be automatically verified using the explicit-state model checking tool DiVinE [2]. The properties to be verified are to be given as formulae of the Linear Temporal Logic (LTL) [13].

In this paper we evaluate the possibilities of DCCL verification by comparing the general DEECo semantics with DCCL and employing DCCL in a real-life case study from the domain of electrical vehicle navigation which comes from one of our industrial partners in the EU FP7 project ASCENS [11]. In particular, we analyze, which verification properties are preserved by the specialization featured by DCCL and which are not. Thus, we identify classes of properties that may be verified by model checking DCCL-based models. Furthermore, we demonstrate how we employed DCCL for verification of the identified property classes on the case study. Finally, we discuss the scalability limits of the verification by providing estimates of the state space size based on the size of the case study problem.

The rest of the paper is organized as follows. In Sect. 2, we describe the case study and articulate the running example that is used throughout the paper. In Sect. 3, we introduce the main concepts of DEECo and illustrate them on the running example. In Sect. 4 we provide a brief overview of the general DEECo semantics, while describing DCCL in Sect. 5. In Sect. 6, we elaborate on the relation between the general DEECo semantics and DCCL. Consequently, in

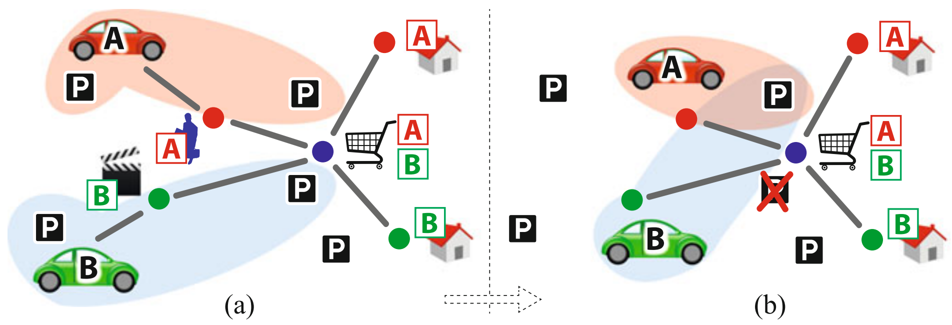


Fig. 1. E-mobility: potential ensembles and their dynamic changes (available parking stations close to respective POIs).

Sect. 7 we demonstrate the DCCL verification on the case study. In Sect. 8 we present a discussion of the experience we have gained while working with the case study. Finally, in Sect. 9 we provide a brief overview of the related work and we conclude the paper in Sect. 10.

2 Case Study

We illustrate the main challenges of EBCS with the help of the electrical vehicle navigation case study – so called e-mobility case study – featured by the ASCENS project, brought to the project by Volkswagen AG [11].

The objective of the e-mobility case study is to coordinate the planning of vehicle journeys in compliance with parking and charging strategies in a highly dynamic and heterogeneous traffic environment, where information is distributed. The case study comprises electric vehicles that have to reach particular Points Of Interest (POIs) within given time constraints. These POIs and their respective constraints are listed in the event calendar of the e-vehicles. E-vehicles are also equipped with sensors of basic capabilities, e.g., monitoring the battery level and energy consumption of the vehicle, but also more sophisticated ones, e.g., monitoring the traffic level along the route. Vehicles can only park and recharge in designated parking spaces and charging lots, organized into parking/charging stations. Vehicles are capable of communicating with each other, as well as parking/charging stations. Such communication is necessary, e.g., in order for a vehicle to obtain the availability of the parking station and potentially reserve a place there. It is important that in this setting no central coordination point is assumed; there is no global control or global planning. Instead, every e-vehicle plans and executes its route individually, based on the data available.

The whole system can be seen as a set of (distributed) nodes, which form ensembles (dynamic communication groups) in order to cooperate on achieving their goal – to allow drivers to arrive at their POIs in time while leveraging the available resources in a close-to-optimal way. This is illustrated in Fig. 1a – each vehicle forms an ensemble with available parking stations close to their respective

POIs. Figure 1b further shows an evolution of the scenario, where vehicles have moved along their planned route and a parking station has become unavailable leading to dynamic changes of the ensembles.

Throughout the paper, we will use a running example that simplifies the e-mobility case study by making the following assumptions: (i) parking and charging stations are modeled together as Parking Lot/Charging Station (PLCS) elements, (ii) vehicles react to changes in the environment only by updating their reserved PLCSs, (iii) availability of PLCSs changes only as a result of reserving a parking place, and (iv) a PLCS will be considered by a vehicle for reservation if it is within a fixed distance to one of the vehicle's POIs. Although simplified, the running example still maintains the important characteristics of the general case study.

We also assume the following conceptual implementation of the running example: (i) each vehicle recurrently aggregates availability information of the relevant PLCSs, e.g., the ones within a fixed distance to one of the vehicle's POIs; (ii) based on this information the vehicle continuously (re-) plans parking/charging periods on a selection of the relevant PLCSs and issues corresponding reservation requests (in the case of re-planning/changes of the selection issues also corresponding cancellation requests); (iii) each PLCS processes its requests and produces confirmations; (iv) having all the reservations confirmed, a vehicle moves towards it's closest destination (while repeating the steps i-iii).

3 DEECo: Key Concepts

Designing a navigation system that targets the case study brings a number of important challenges. In particular: (i) the physical architecture of the system constantly changes as the vehicles/PLCSs might enter/leave the system at any point (e.g., due to low connectivity or physical unavailability); (ii) vehicles and PLCSs have only a partial view over the whole system, according to the information they obtain from components they interact with; and (iii) the trip planning and decision making is decentralized and localized to the vehicles. In this section we illustrate the key concepts of the DEECo component model – a representative of EBCS – on the running example and demonstrate how the challenges listed above are addressed using these concepts. A DEECo-specific implementation of the running example is outlined in Fig. 2 and Fig. 3.

As illustrated in Fig. 2, a component (e.g., lines 7–20) comprises knowledge and processes. Knowledge (lines 8–9) represents the internal data of the component; it can be exposed to the rest of the system via the component's interfaces (e.g., lines 1–2, 7). A process (lines 10–20) is essentially a thread operating upon the component knowledge in a cyclic manner (similar to a feedback loop). For example, `Vehicle0123` in Fig. 2 is a component, in which the `move` process updates the vehicle's next position based on its current position, the route calendar, and the current reservation status. The process is executed periodically every 100 ms. An important restriction of component processes that facilitates autonomy and resilience, is that there is no direct communication (i.e., remote method invocation or message exchange) among components in the system. Each component

```
interface Vehicle:
    calendar, availabilityList

interface PLCS:
    position, availability

component Vehicle0123 features Vehicle:
    knowledge:
        id, position, calendar, calendarFeasibility, availableParkingLots, reservations, cancellations
    process computeReservations(in calendar, in availabilityList, inout reservations, inout cancellations):
        function:
            oldReservations ← reservations
            reservations ← selectParkingLotsToBeReserved(calendar, availabilityList, oldReservations)
            cancellations ← determineReservationsToBeCancelled(oldReservations, reservations)
        scheduling: triggered( changed(availabilityList) )
    process move(inout position, in calendar, in reservations)
        function:
            if (allPOIsReserved(calendar, reservations))
                position ← moveToNextPosition(position, calendar)
        scheduling: periodic( 100ms )
    ...

component PLCS01 features ParkingLot:
    knowledge:
        id, position, availability, reservations
    process processReservations(inout availability, inout reservations):
        function:
            availability ← reserveFreePlaces(availability, reservations)
            reservations ← markProcessedReservations(availability, reservations)
        scheduling: periodic( 2000ms )
    ...
```

Fig. 2. Example of DEECo components in a DSL

operates solely upon its own knowledge. Nevertheless, a component's knowledge may include it's belief about the knowledge of other components. Updates of this belief are completely externalized into component ensembles, described below.

As illustrated in Fig. 3, an ensemble (e.g., lines 32–40) is a first-class concept that enables dynamic grouping of components and interaction between the components in the group. A component in an ensemble assumes either the role of the unique ensemble coordinator, or the role of one of the potentially multiple members. The role of a component is determined dynamically by the membership condition (lines 35–37) over component interfaces (lines 33–34). For example, PropagateReservationRequests in Fig. 3 is an ensemble, in which a Vehicle and a PLCS form a coordinator-member pair if the Vehicle's reservations include the PLCS. Technically, the run-time platform is responsible for timely evaluation of the condition. As indicated above, the only mechanism for component interaction is updating the interacting components' belief. This is done via the knowledge exchange of an ensemble (lines 38–40). Similar to component processes, knowledge exchange is a cyclic activity that updates the coordinator's belief about the members and vice versa. For example, in PropagateReservationRequests the knowledge exchange updates every 5000 ms the belief of member PLCSs about the relevant reservations of the coordinating Vehicle. Again, the run-time platform is responsible for timely knowledge exchange execution among all components that are in the same ensemble.

```
ensemble UpdateAvailabilityInformation:
  coordinator: Vehicle
  member: PLCS
  membership:
    ∃ poi ∈ coordinator.calendar:
      distance(member.position, poi.position) ≤ TRESHOLD
  knowledge exchange:
    coordinator.availabilityList ← reduce(member.availability)
  scheduling: periodic( 5000ms )

ensemble PropagateReservationRequests:
  coordinator: Vehicle
  member: PLCS
  membership:
    ∃ reservation ∈ coordinator.reservations:
      reservation.plcsId == member.id ∧ reservation.status == NEW
  knowledge exchange:
    member.reservations.add(coordinator.reservations.getNewForPLCS(member.id))
  scheduling: periodic( 5000ms )

ensemble PropagateReservationConfirmations:
  ... // similar to the previous, opposite direction
```

Fig. 3. Example of DEECo ensembles in a DSL

4 General DEECo Semantics

DEECo comes with a well-defined general semantics, which faithfully captures the operation of a DEECo-based application and its run-time platform in real environment by accounting for (a) fully asynchronous, distributed, and decentralized operation of components and ensembles, (b) real-time scheduling, and (c) network specific issues such as communication delays and losses. In this section we describe the general semantics, because it establishes a baseline for verification of DEECo-based applications. Other DEECo semantics aimed at verification (e.g., DCCL) or stemming from implementations of DEECo by employing different communication middleware are further seen as specializations of the general semantics – meaning that they generate only a subset of execution traces permitted by the general semantics.

The general DEECo semantics describes a DEECo-based application as a finite-state non-deterministic automaton, whose states capture the knowledge of the system's components and the transitions correspond to execution of component processes or ensemble knowledge exchange. Note, that although the general DEECo semantics could support infinite knowledge domains, we consider only finite domains. This poses no real limitation, since typical CPS applications are limited in terms of available resources. In particular, we construct the automaton as a Cartesian product of three groups of automata pertaining to: (i) component processes, (ii) knowledge propagation, and (iii) ensemble knowledge exchange.

4.1 Component Processes

A component process is an activity local to a component that atomically reads a subset of the component's knowledge, performs computation on it (possibly performing sensing and actuation), and atomically updates the component's knowledge with the result of the computation. To model this, we associate each process p of each component C with an automaton $A(p)$ – depicted in Fig. 4. The initial state of the automaton is the *idle* state. The transition p_1 corresponds to reading the component knowledge (denoted V_C) into a temporary variable. The transition p_2 reflects both the execution of the process and updating the component's knowledge with the outcome. Such semantics allows for concurrent, asynchronous execution of component processes.

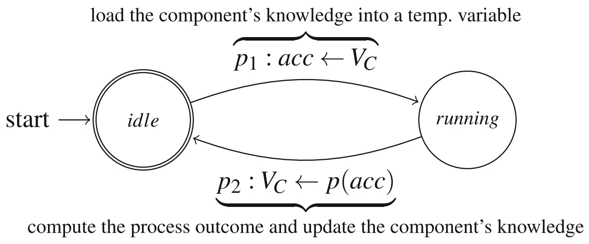

load the component's knowledge into a temp. variable
$p_1 : acc \leftarrow V_C$
start
idle
running
$p_2 : V_C \leftarrow p(acc)$
compute the process outcome and update the component's knowledge

Fig. 4. Component process automaton – $A(p)$

4.2 Knowledge Propagation

As mentioned earlier, components in a DEECo system can only interact via knowledge exchange prescribed by ensemble definitions and realized by the runtime platform. The particulars of distributed communication required to realize knowledge exchange very much depend on the communication middleware used. To keep the execution semantics sufficiently general, we model the distributed communication with relatively few restrictions. In particular, we assume that each component C is associated with an arbitrarily outdated copy of knowledge valuation of any other component C' in the system – the so called *C's view of C'* (denoted as $V_C^{C'}$). Note, that the concept of view is different from belief (the former being a technical means of defining the semantics, the latter expressing the application-specific purpose of a part of component knowledge).

To capture knowledge propagation in terms of updates of component views, we associate a queue $Q_{C_i}^{C_j}$ with each ordered pair of components $C_i, C_j, C_i \neq C_j$, which serves as a communication channel for the knowledge valuations of C_j that are being propagated through the network to become the C_i's view of C_j ($V_{C_i}^{C_j}$). We assume the queue to be an unbounded perfect FIFO queue without errors.

As depicted in Fig. 5, in order to model the actions of knowledge propagation and propagation delays associated with sending and receiving the knowledge valuations over the network, we associate with each queue $Q_{C_i}^{C_j}$ an automaton ($A(Q_{C_i}^{C_j})$). The transition q_1 of this automaton corresponds to sending the knowledge valuation of C_i to C_j in terms of putting it into the queue. In a similar manner, the transition q_2 corresponds to updating C_i's view of C_j ($V_{C_i}^{C_j}$) in terms of retrieving it from the queue.

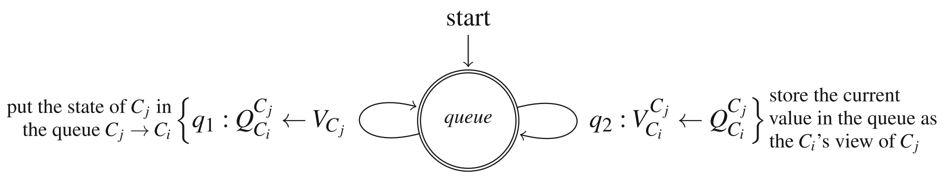

start
queue
put the state of C_j in the queue $C_j \to C_i$
$q_1 : Q^{C_j}_{C_i} \leftarrow V_{C_j}$
$q_2 : V^{C_j}_{C_i} \leftarrow Q^{C_j}_{C_i}$
store the current value in the queue as the C_i's view of C_j

Fig. 5. Knowledge propagation automaton – $A(Q^{C_j}_{C_i})$

Note that the mandatory association of such view with each component is only needed for the definition of semantics. The DEECo run-time framework does not provide a corresponding run-time concept – it only provides a general contract regarding the general spread of component knowledge valuations throughout the system, without any specific guarantees.

4.3 Ensemble Knowledge Exchange

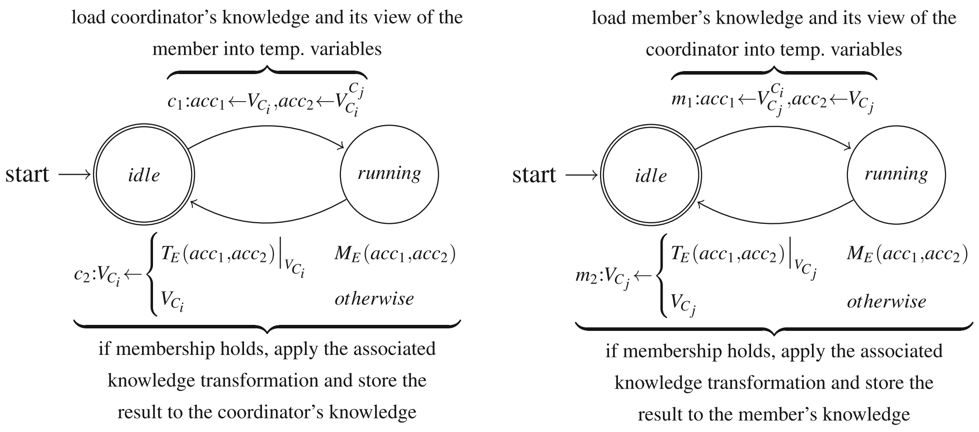
In an ensemble the knowledge exchange takes place always between the coordinator and the members. For the sake of simplicity, in the definition of the semantics, we treat an ensemble as a set of binary relations between a single coordinator and each of the corresponding members.

Note that while the general propagation of knowledge throughout the (distributed) system, modeled via queues, concerns the whole knowledge of the involved components, the ensemble knowledge exchange concerns only certain knowledge fields, specific for the ensemble.

To capture the asynchrony and dependence on knowledge propagation, the knowledge exchange is modeled as a set of component-specific automata locally manipulating the component's knowledge and views.

In particular, as depicted in Fig. 6a, we associate the role of the coordinator (C_i) of an ensemble with an automaton $A_c(E^{C_j}_{C_i})$. Similarly to the process automaton, the process of loading and processing the knowledge is divided into two states – *idle* and *running*, modeling thus asynchronous processing. The transition c_1 corresponds to loading the coordinator's knowledge and it's view of one of the members into temporary variables. The transition c_2 then reflects the storing of the outcome of the knowledge exchange, i.e., the effect of the knowledge transformation T_E associated with the knowledge exchange applied on the temporary variables, in the case the ensemble membership (M_E) holds.

Similarly, we associate the role of a member C_j of the ensemble with an automaton $A_m(E^{C_j}_{C_i})$, as depicted in Fig. 6b. The automaton is very similar to the one in Fig. 6a, with the difference that the member's knowledge and view of the coordinator are interchanged in both T_E and M_E (i.e., C_i and C_j switched the roles in the automaton).


load coordinator's knowledge and its view of the member into temp. variables
$c_1{:}acc_1 \leftarrow V_{C_i}, acc_2 \leftarrow V_{C_i}^{C_j}$
start
idle
running
$c_2{:}V_{C_i} \leftarrow \begin{cases} T_E(acc_1, acc_2)\big|_{V_{C_i}} & M_E(acc_1, acc_2) \\ V_{C_i} & otherwise \end{cases}$
if membership holds, apply the associated knowledge transformation and store the result to the coordinator's knowledge
load member's knowledge and its view of the coordinator into temp. variables
$m_1{:}acc_1 \leftarrow V_{C_j}^{C_i}, acc_2 \leftarrow V_{C_j}$
start
idle
running
$m_2{:}V_{C_j} \leftarrow \begin{cases} T_E(acc_1, acc_2)\big|_{V_{C_j}} & M_E(acc_1, acc_2) \\ V_{C_j} & otherwise \end{cases}$
if membership holds, apply the associated knowledge transformation and store the result to the member's knowledge

(a) Coordinator – $A_c(E_{C_i}^{C_j})$ (b) Member – $A_m(E_{C_i}^{C_j})$

Fig. 6. Ensemble knowledge exchange automaton

4.4 System Semantics

Building on the previously introduced specific automata, we can now define the semantics of a system S consisting of a set of components $\mathbb{C}$, each of which including a set of processes $\mathbb{P}_C$, and a set of ensemble definitions $\mathbb{E}$ via the following automaton:

$$A(S) = \underbrace{\prod_{C \in \mathbb{C}} \prod_{p \in \mathbb{P}_C} A(p)}_{\substack{\text{processes of all} \\ \text{components}}} \times \underbrace{\prod_{\substack{C_i, C_j \in \mathbb{C} \\ C_i \neq C_j}} A(Q_{C_i}^{C_j})}_{\substack{\text{knowledge propagation} \\ \text{between each two} \\ \text{components}}} \times \underbrace{\prod_{E \in \mathbb{E}} \prod_{\substack{C_i, C_j \in \mathbb{C} \\ C_i \neq C_j}} \Big(A_c(E_{C_i}^{C_j}) \times A_m(E_{C_i}^{C_j}) \Big)}_{\substack{\text{knowledge exchange between each two} \\ \text{components and for each ensemble}}}$$

As already indicated in the automaton definition, a system automaton aggregates automata for all the processes of all the components. To capture all the potential ensembles among the components, it also includes a knowledge propagation automaton between each oriented pair of components, as well as both coordinator and member automata for each ensemble definition and each oriented pair of components; i.e., each two components can potentially form a coordinator-member pair of an ensemble. Being completely non-deterministic, the system automaton can also capture system behaviors, that are not realistic w.r.t. real system execution. Therefore, we impose further restrictions on the system automaton in terms of limiting its set of valid execution traces. In particular, as DEECo and EBCS systems in general are soft realtime cyber-physical systems, we focus on realtime properties of the execution traces.

In principle, we allow only those traces of the system automaton that are realistic with respect to a soft-realtime periodic scheduling of the included process/propagation/exchange automata. Namely, we impose the following restrictions:

- Given the duration of the period of a component process, the process has to start and end within each period (i.e., each period the corresponding process automaton has to go from the *idle* to *running* state and back).
- Given the duration of the period and the maximum expected network latency, all knowledge propagation has to be performed within each period with the maximum delay equal to the latency (i.e., each period the corresponding knowledge has to be enqueued, while it is dequeued with a delay at most equal to the latency)
- Similarly to a component process, given the duration of the period, all knowledge exchange has to start and end within each period (i.e., each period all the corresponding coordinator and member automata have to go from the *idle* to *running* state and back).

In a way, these restrictions impose specific "fairness" constraints on the system automaton traces that are brought about by the properties of the run-time platform. Since the technical details are beyond the scope of this paper, we refer an interested reader to [1].

5 DCCL: Semantics Suitable for Verification

Due to the extremely big state space the general DEECo semantics generates, it is not suitable for verification. To accomplish the verification task, we have developed DCCL, which acts as a specialization of the general DEECo semantics that is suitable for LTL model-checking using the model checking tool DiVinE [2].

Compared to the general DEECo semantics, DCCL incorporates certain simplifications to keep the state space reasonably small and the model-checking task thus tractable. In particular, DCCL specializes the general semantics by omitting component views and assuming knowledge propagation to be instant. Furthermore, DCCL restricts the syntactic expressiveness of DEECo in the following way. It allows only one process per component. The set of possible knowledge of a component has to be finite, i.e., we restrict the data of each component to be variables over a finite domain. As already mentioned, this restriction poses no real limitation for typical CPS applications. All processes in the system have to be periodic, synchronously activated and they strictly alternate with knowledge exchange, which is also synchronously activated. We outline the DCCL semantics below, for more details, we refer the reader to [3].

The computation of a DCCL system works in two alternating phases, the *component phase* and the *ensemble phase*. In the component phase, components perform their computation as prescribed by their process description. After the component phase, the system switches to ensemble phase, where the ensemble knowledge exchange is performed. In order to capture various kinds of timing constraints, we provide two different kinds of semantics for the ensemble phase. The first, *fixpoint semantics*, represents a situation in which the knowledge exchange is infinitely faster than the progress of the components' processes, i.e., it takes negligible time. In this semantics, the ensemble phase proceeds as long as there is some knowledge exchange to be done. The second semantics, *timeunit semantics*, is assigned a single number, a time limit ℓ. The ensemble phase then proceeds as in the previous, this time respecting the fact that every component may only take part in as many as ℓ knowledge exchanges.

Formally, we define a labeled transition system $(\Sigma, \mathcal{L}, \rightarrow)$, where Σ is a set of states, $\mathcal{L}$ is a set of labels and $\rightarrow \subseteq \Sigma \times \mathcal{L} \times \Sigma$ is a labeled transition relation. The definition of states depends on the desired semantics. In the fixpoint semantics, a state consists of the knowledge of every component and a marker indicating the current phase.

$$\Sigma_{\mathrm{f}} = K_{C_1} \times \cdots \times K_{C_n} \times \{\mathbf{C}, \mathbf{E}\}$$

In the timeunit semantics, each component includes a time counter whose maximal value is ℓ, the time limit. The ensemble phase marker is also enhanced with a similar counter, called the *round.*

$$\Sigma_\ell = K_{C_1} \times \{0, \ldots, \ell\} \times \cdots \times K_{C_n} \times \{0, \ldots, \ell\} \times (\{\mathbf{C}\} \cup (\{\mathbf{E}\} \times \{1, \ldots, \ell\}))$$

5.1 Component Phase

The progress of the component phase is very straightforward. Each component only possesses a single process and the processes are independent, as they may not touch other components' knowledge. We may thus perform all processes synchronously at once. Formally, whenever σ is a state of our transition system containing the marker $\mathbf{C}$, let σ' denote its modification as follows: All components' knowledge in σ is changed according to the components' processes, the marker is changed to $\mathbf{E}$, and, if the semantics is timeunit, all component time counters as well as the round counter are set to ℓ. We then have the transition $\sigma \xrightarrow{comp} \sigma'$.

5.2 Ensemble Phase

The ensemble phase consists of a number of smaller units, the knowledge exchange steps. Every such step performs the knowledge exchange between one coordinator and one member. The number of steps performed in each ensemble phase depends on the chosen semantics. In a sense, the choice of semantics thus governs the amount of fairness that is provided in the ensemble phase.

Fixpoint Semantics. In the fixpoint semantics, the ensemble phase runs until a fixpoint is reached, i.e., until no more knowledge exchange steps can be performed. Formally, let σ be a state of the transition system containing the ensemble phase marker $\mathbf{E}$. Let us consider all possible combinations of an ensemble

E with membership predicate p and knowledge exchange e, and two components C_i, C_j such that $p(C_i, C_j)$ holds in σ. This means that C_i is currently a coordinator of an ensemble and that C_j is one of its members. For every such triple (E, C_i, C_j), let σ' be the state that is created from σ by changing the knowledge of C_i, C_j according to the knowledge exchange e. We then have the transition $\sigma \xrightarrow{ens(E,C_i,C_j)} \sigma'$. Note that as there might be more triples satisfying the properties above, the evolution of state σ is possibly non-deterministic.

If there is no possible knowledge exchange in the state σ, i.e., no transition from σ has been created according to the above rule, the ensemble phase ends. We represent this by a transition $\sigma \xrightarrow{end} \sigma'$ where σ' is equal to σ with the phase marker changed to **C**.

Timeunit Semantics. In the timeunit semantics, the number of steps of the ensemble phase is limited with a given number ℓ so that every component takes part in at most ℓ knowledge exchanges during the phase. At the same time, we want to perform as many exchanges as possible. We thus divide the ensemble phase into ℓ rounds, numbered ℓ, $\ell - 1$, ..., 1. In round k, knowledge exchange only occurs if both participants still have k time units left.

Formally, let σ be a state with phase marker **E** and its round counter set to k. Let (E, C_i, C_j) satisfy the same property as in the previous semantics with the additional constraint that the time counters of both C_i and C_j are set to k. For every such triple, let σ' be the state that is created from σ by changing the knowledge of C_i, C_j and lowering their time counters to $k-1$. We then have the transition $\sigma \xrightarrow{ens_k(E,C_i,C_j)} \sigma'$.

Again, if there is no possible exchange in the state σ, the round ends. If $k > 1$, the next round starts. The state σ' is created from σ by changing the round counter, as well as all time counters that have still k units left, to $k - 1$ and we have the transition $\sigma \xrightarrow{round} \sigma'$. If $k = 1$, the ensemble phase ends. The state σ' is created from σ by changing all time counters to zero and changing the phase marker to **C**. We then have $\sigma \xrightarrow{end} \sigma'$.

6 Relation of DCCL to the General DEECo Semantics

Since DCCL is a simplification of the general DEECo semantics (e.g., while in DEECo processes are fully parallel, in DCCL they are executed synchronously), it is natural that model-checking of a DCCL-based system cannot verify all potential properties that could be expressed over the traces of the general semantics. In this section we thus discuss the relation of the general DEECo semantics and DCCL with respect to analyzable properties. In particular, we identify (i) properties which have equivalent validity under DCCL and general semantics, and (ii) properties which do no have equivalent validity (i.e., they pertain to aspects of the general semantics that are abstracted away by DCCL).

Note also that DCCL is a specialization of the general semantics, which means that DCCL semantics cannot introduce a trace the equivalent of which could not be produced by the general DEECo semantics. (Here, we consider two traces equivalent if they entail the same sequence of knowledge updates.) This means that violation of a property under DCCL implies violation of an equivalent property under the general semantics.

6.1 Realistic Properties That Can Be Verified via DCCL

Based on LTL model-checking procedure [13], we can essentially verify the traditional properties such as safety or liveness for a DCCL-based system, where we consider the model to be an implementation of the system in the DCCL language. Specifically, in our approach the atomic propositions of LTL specification range over component knowledge valuations. In the following, we discuss in more details a classification of properties specific to DEECo concepts that can be verified for DCCL-based models.

Correctness of process execution (P1). Since DCCL explicitly represents the state of a component's knowledge before and after process execution, we can effectively exploit the LTL property checking to verify correctness of the process execution. This is naturally an important concern in DEECo.

Correctness of component interaction protocols (P2). Building on (P1), we can also verify execution of a sequence of component processes and knowledge exchange. This enables us to verify correctness of interaction protocols between components (embodied by the sequence). In DEECo, due to the specifics of the development cycle [4], the correctness of interaction is an important concern as both component processes and ensemble knowledge exchange are developed in isolation and the component interfaces do not provide enough semantic information.

Resilience w.r.t. knowledge inconsistency (P3). In this case, we want to verify that a given system is resilient w.r.t. knowledge inconsistency caused by parallelism of knowledge exchange/component processes (i.e., a component receives up-to-date information from one component while receiving stale information from another, because at the time the information was sent the other component's process has not yet produced the new information). Also, the inconsistencies can be introduced due to variable communication delays. This is an important concern under the general DEECo semantics, since the semantics allows complete parallelism of processes and knowledge exchange (limited only by the realtime constraints) and it imposes little constraints on the communication delays. Although in general the DCCL semantics does not implicitly account for such parallelism/delays, in order to reduce the complexity of a model and its verification, it is possible to capture the parallelism/delays in DCCL explicitly. Specifically, this is done by enabling an execution of a process/knowledge exchange to be non-deterministically skipped, while enforcing fairness (i.e., delayed for a finite number of periods). Technically, this can be done by introducing a specific flag into the relevant components' knowledge and defining an "artificial" ensemble, that manipulates the flag. The other ensembles/processes have to be then modified in such a way, that they will do nothing if the flag is set. The non-deterministic interleaving of knowledge exchange during ensemble phase will then yield two branches (depending, whether the artificial ensemble was evaluated before the others), one where the flag has been set and the corresponding original ensembles/processes did not have any effect (i.e., was delayed), and one where the ensembles/processes proceeded as before, thus simulating non-deterministic delays.

Communication-boundedness of interaction protocols (P4). Building on the timed semantics of DCCL, we can verify whether a particular interaction is communication-bounded – i.e., whether its correctness depends on a particular communication speed. Specifically, we can verify that the system will manifest erroneous behavior if the time limit of the timeunit semantics is too low, while otherwise behaving correctly. Such a situation is realistic w.r.t. the general DEECo semantics, since the semantics does not provide any specific guarantees for the knowledge propagation and network latency. This concern can be critical for certain application that require a high-level of safety and dependability (i.e., some behavior should be correct under arbitrary communication conditions).

6.2 Realistic Properties That Cannot Be Verified via DCCL

Properties related to parallelism of processes and knowledge exchange. Being virtually synchronous, the DCCL semantics does not explicitly allow to verify properties based on parallelism of processes and knowledge exchange, such as race conditions (as allowed by the DEECo semantics). To partially remedy this problem, we can modify the DCCL model so that an execution of a process/knowledge exchange can be non-deterministically skipped, which is the case of (P3). Nevertheless, this still does not reflect complete parallelism.

Properties related to real time. Expanding on the previous point, the DCCL semantics does not allow verification of properties related to real time execution, such as that the periods of processes and knowledge propagation/exchange that have mutual knowledge dependencies are set up correctly, i.e., so that they together provide a satisfactory end-to-end response time. Similar to the previous case, DCCL allows only for a partial solution based on a simple discretization of time, which is the case of (P4).

7 Modeling the Case Study

In order to evaluate DCCL w.r.t. verification of realistic DEECo properties (Sect. 6.1), we have fully modeled the running example presented in Sect. 2, while following the implementation outlined in Sect. 3. Note that the running example retains many important challenges of the realistic, industry-relevant case study, rather than being a mere experimental setting.

Naturally, DCCL introduces a large amount of abstraction w.r.t. the original system behavior. For instance, the time intervals of parking reservations have been discretized into a finite set of time "slots". Similarly, we have also discretized the geographic positions and distance. Moreover, the model includes only a simple and fully deterministic implementation of algorithms for planning vehicle routes, deciding PLCSs for parking, as well as for assigning parking places to vehicles.

To gain a better insight in the impact of concurrency in knowledge exchange, we have modeled two different variants of knowledge exchange of parking requests in the corresponding ensembles. Specifically, these variants are concerned with ordering of the requests coming from multiple parties concurrently. In the first variant, which we will call "standard", the requests are ordered based on their content, thus eliminating the impact of concurrency. In the second variant, which we will call "first-come-first-served", the requests are ordered according to the order in which knowledge exchange was executed, thus emulating the first come first served semantics of message queues.

Note that although we always select particular components/PLCSs in the following examples, in our experiments all the vehicles/PLCSs were symmetric so that the selection of a particular one does not corrupt the generality of the example.

7.1 Verification of Realistic Properties on the Case Study

To illustrate the potential of verifying realistic properties using DCCL on the case study, we have formulated and verified at least one property of each class identified in Sect. 6.1.

Correctness of process execution (P1). As an instance of a (P1) property, we have checked that the process of PLCS, assigning parking places to vehicles, is correct. We have done it by verifying that a PLCS never assigns a single parking space to two vehicles for the same time slot. This property can be expressed via the following LTL formula: G(!v0_assigned_the_same_place_as_v1). The atomic proposition v0_assigned_the_same_place_as_v1 checks in a straightforward way the knowledge of each PLCS, its buffer storing the processed reservation requests in particular, whether the two vehicles (i.e., 0 and 1) have been assigned the same parking place. As an aside, using this property we have been able to localize an error in the parking-place-assignment process of PLCS, which was based on not marking a parking place as occupied after assigning it to a vehicle, and thus assigning it twice.

Correctness of component interaction protocols (P2). As for the (P2) property class, we have verified that whenever a vehicle creates a reservation request, the vehicle gets eventually notified about confirmation or rejection of the request by the corresponding PLCS. This is expressed by the formula G(v0_requests_p0 -> (v0_requests_p0 U v0_request_to_p0_decided)) (again, for convenience we have used a fixed pair vehicle-PLCS). Here, v0_requests_p0 is true whenever the vehicle 0 contains a new parking request for PLCS 0, while v0_request_to_p0_decided is true whenever the vehicle knows the decision of PLCS

0 on its request (be it either confirmation or rejection). Both atomic propositions are simple checks on the knowledge of vehicle 0. This property was verified under the assumption that a PLCS's knowledge can accommodate requests of all relevant Vehicles. Recall that each Vehicle produces a single request for each of its calendar events and waits for the decision.
Resilience w.r.t. interaction inconsistency/delays (P3). Using a (P3) property, we have verified that the system is resilient w.r.t. inconsistency of PLCS-availability information in a vehicle. Specifically, we have done it by verifying the property that a vehicle's reservations are always valid even if its PLCS-availability information is inconsistent due to delays in communication. This is done by verifying the formula G(v0_has_confirmation_for_p0_t0 -> p0_blocks_place_for_v0_t0) in a modified model where the exchange of PLCS availability can be non-deterministically delayed for one ensemble phase. Here, v0_has_confirmation_for_p0_t0 is true whenever vehicle 0 has a confirmed reservation of a place on PLCS 0 for the time slot 0, while p0_blocks_place_for_v0_t0 is true whenever PLCS 0 blocks a parking place for vehicle 0 for the same time slot. Both atomic propositions are simple checks over the corresponding vehicle/PLCS knowledge. The non-determinism is implemented as indicated in Sect. 6.

Communication-boundedness of interaction protocols (P4). As to the class (P4), we have tried to assess the communication-boundedness of the reservation request interaction protocol. For this, we have exploited the property that we have used to illustrate (P2) – G(v0_requests_p0 -> (v0_requests_p0 U v0_request_to_p0_decided)). Under the timeunit semantics, since it limits the number of interactions allowed in a single ensemble phase, the property will not longer hold (for sufficiently small time limits). Thus the interaction protocol concerned with exchanging parking reservation requests is communication bounded. Knowing this, we could improve the design of the vehicle component so that it is resilient w.r.t. this situation. Technically, we have done it by keeping the vehicle idle until it receives a confirmed reservation for all its requests. Note that for example the PLCS-availability exchange protocol is not communication-bounded, as the vehicle does not distinguish whether the availability information is missing because the PLCS is not relevant to any of its POIs or whether it just did not get through due to slow communication.

7.2 Scalability Evaluation

To evaluate the scalability of DCCL w.r.t. the case study, we have measured the size of the state space for different configurations of the case study (i.e., different numbers and initial states of vehicles/PLCSs). Specifically, to obtain comparable results, the configurations enforce the maximum amount of successful interaction expected for the given number of components (i.e., without parking request conflicts).

The scalability of DCCL is illustrated in Fig. 7. As expected, the number of states grows exponentially w.r.t. the number of components in the system. The curve of growth is relatively steep, however, this is acceptable given the complexity of the case study and therefore also the corresponding model. Naturally, the

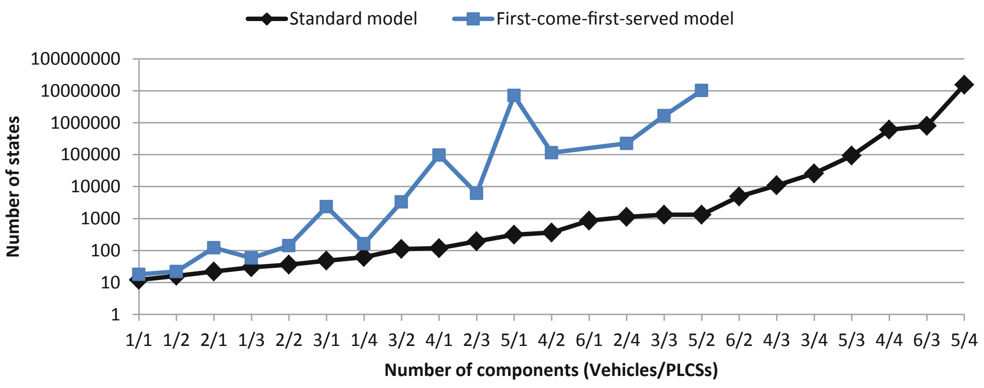


Fig. 7. Scalability of DCCL w.r.t. the case study

first-come-first-served variant of knowledge exchange scales much worse than the standard variant, since it generates much more states at the end of each ensemble phase (capturing different permutations of requests exchanged during the phase). In a similar way, the same configurations yielded a much bigger state space under the timeunit semantics, since the time-constrained prefixes of knowledge exchange sequences produced a lot more different states at the end of each ensemble phase. Since the size of the state space depends on the time limit in a complex way, we have not included this variant in the figure.

8 Discussion

8.1 Lessons Learned

A major asset of our approach to verification of DEECo-based applications is that DCCL is based on the DiVinE model checker, which is a mature, reliable, and well-performing tool with solid supporting infrastructure. This helped especially when verifying a large model including non-trivial behavior, such as the one modeling the case study.

Our experiments show that even after introducing relatively significant simplifications to the execution model (such as synchronous alternation of component/ensemble phases, unlimited time for knowledge exchange under the fixpoint semantics), it is still possible to maintain a rich set of verifiable properties. Specifically, this observation appears to apply not only to DEECo-based systems, but also to EBCS and even cyber-physical systems in general, since they share common basic characteristics. Nevertheless, there are still some aspects of DCCL, such as no explicit support for non-determinism in component processes/knowledge exchange, that introduce unnecessary complexity and thus could be targeted in the future work.

Finally, we argue that when modeling non-trivial examples, the organization of data within the model has a significant impact both on the size of the state space (e.g., fixed index assignment vs. first-come-first-served), as well as in terms of complexity (e.g., regarding formulation of atomic propositions and LTL formulae). However, this issue has been addressed little in the contemporary model-checking approaches and thus further investigation of this topic would be beneficial (for instance by providing guidelines).

8.2 Improving Scalability by Ensemble State Reduction

As shown in Sect. 7, DCCL is not yet able to scale to bigger configurations. This can be partially remedied by employing a specific state-space reduction during ensemble phase. In particular, since none of the properties that we have experimented with relied on a valuation of atomic propositions in an internal ensemble-phase state (i.e., a state that has transitions only from/to states in the same ensemble phase), it should be possible to reduce the state space by eliminating these internal ensemble-phase states. Technically, this can be done by discarding the internal states at the end of each ensemble phase, after all the final states of the phase have been generated.

Nevertheless, as also shown in Sect. 7, the current level of scale still suffices to verify important realistic properties of a modeled system. Also, as supported by our experiments, arguably a large number of important property violations can be detected early, on a reasonably small configuration.

9 Related Work

As to the general domain of EBCS, we are currently not aware of any other approach that would be directly related to DEECo and DCCL. However, there is a number of approaches targeting similar domains; i.e., similar to CPS.

Closest to the area of EBCS, SCEL [8] is targeting a formalization of the semantics of attribute-based communication (i.e., the key concept behind EBCS ensembles) in the domain of formal coordination languages, with the future intention of exploiting the formal semantics for analysis and verification, as well as evaluation on an extensive case study.

When considering the broader domain of real-time embedded systems, which share a number of aspects of CPS, there exists a number of approaches for verification of safety and timing properties. These include well-established languages such as AADL[1], EAST-ADL[2], and VERDE/MARTE[3], which come with a number of related tools (e.g., COMPASS[4]) mainly focusing on timing and dependability analysis, or CHESS[5] methodology and toolset mainly focusing on timing, failure propagation and dependability analysis. The closest to our model-checking of EBCS is the approach of OTHELLO/OCRA [5], which allows for

[1] http://www.aadl.info
[2] http://www.east-adl.info
[3] http://www.itea-verde.org
[4] http://compass.informatik.rwth-aachen.de
[5] http://www.chess-project.org

checking of refinement of contracts expressed in a variant of linear-time temporal logics interpreted over hybrid traces (i.e., traces that contain both discrete events and continuous-time state evolution). Although all these approaches and tools target a closely related domain to DCCL, they require a significant shift from the EBCS concepts, thus increasing the effort required for modeling and reducing the value of the verification results.

Our technique of model checking DCCL is built on top of the parallel and distributed explicit-state model checker DiVinE [2]. DiVinE primarily offers the verification of LTL properties by means of the automata-based LTL model checking [13]. DiVinE accepts various input formats, one of them being the binary Common Explicit-State Model Interface (CESMI), which we use for DCCL verification. The translation of a DCCL input file into a CESMI-compliant module is provided via our tool `dccl2cesmi`[6].

10 Conclusion

In this paper, we have discussed the verification possibilities of the DEECo component model, a representative of Ensemble-Based Component Systems (EBCS). In order to make the verification task feasible, we have designed a syntactic and semantic specialization of DEECo called DCCL, verification of which is based on an explicit-state model checker – DiVinE. We have further evaluated the possibilities of DCCL verification on a real-life case study and we have discussed its limitations.

In the future, we would like to focus on two areas. One is that of further extending DCCL to capture more relevant aspects of DEECo, e.g., introducing specific data structures for knowledge-exchange-related tasks. The other area is then that of reducing the state space. DiVinE itself performs certain generic reductions, such as partial order reduction. However, we want to try reductions that are specific to DCCL, such as some kind of symmetry reduction or the reduction of the ensemble steps. In a more distant future, we would like to extend the DCCL verification with quantitative aspect such as probability or precise timing constraints.

Acknowledgments. This work has been supported by the Czech Science Foundation grant project no. P202/11/0312.

References

1. Al Ali, R., Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: DEECo computational model-I., Technical Report D3S-TR-2013-01, D3S, Charles University in Prague. http://d3s.mff.cuni.cz/publications (2013)
2. Barnat, J., et al.: DiVinE 3.0 – an explicit-state model checker for multithreaded C & C++ programs. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 863–868. Springer, Heidelberg (2013)

[6] http://paradise.fi.muni.cz/dccl/

3. Barnat, J., Beneš, N., Černá, I., Petruchová, Z.: DCCL: verification of component systems with ensembles. In: Proceedings of CBSE '13. pp. 43–52. ACM, New York (2013)
4. Bures, T., et al.: DEECo - an ensemble-based component system. In: Proceedings of CBSE '13. ACM, New York (2013)
5. Cimatti, A., Tonetta, S.: A property-based proof system for contract-based design. In: Proceedings of SEAA 2012. IEEE CS, Los Alamitos (2012)
6. Hölzl, M., Rauschmayer, A., Wirsing, M.: Engineering of software-intensive systems: state of the art and research challenges. In: Wirsing, M., Banâtre, J.-P., Hölzl, M., Rauschmayer, A. (eds.) SoftWare-Intensive Systems. LNCS, vol. 5380, pp. 1–44. Springer, Heidelberg (2008)
7. Keznikl, J., et al.: Towards dependable emergent ensembles of components: the DEECo component model. In: Proceedings of WICSA/ECSA'12. IEEE (2012)
8. De Nicola, R., Ferrari, G., Loreti, M., Pugliese, R.: A Language-Based Approach to Autonomic Computing. In: Beckert, B., Bonsangue, M.M. (eds.) FMCO 2011. LNCS, vol. 7542, pp. 25–48. Springer, Heidelberg (2012)
9. Patikirikorala, T., Colman, A., Han, J., Wang, L.: A systematic survey on the design of self-adaptive software systems using control engineering approaches. In: Proceedings of SEAMS 2012 (2012)
10. Rajkumar, R.R., Lee, I., Sha, L., Stankovic, J.: Cyber-physical systems: the next computing revolution. In: Proceedings of DAC'10. pp. 731–736. ACM, New York (2010)
11. Serbedzija, N., Reiter, S., Ahrens, M., Velasco, J., Pinciroli, C., Hoch, N., Werther, B.: Requirement specification and scenario description of the ascens case studies (2011), deliverable D7.1. http://www.ascens-ist.eu/deliverables
12. Shoham, Y., Leyton-Brown, K.: Multiagent Systems: Algorithmic, Game-theoretic, and Logical Foundations. Cambridge University Press, Cambridge (2009)
13. Vardi, M., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: Proceedings, Symposium on Logic in Computer Science (LICS'86), pp. 332–344. IEEE Computer Society (1986)

3.6 Adaptive Deployment in Ad-Hoc Systems Using Emergent Component Ensembles: Vision Paper

Lubomír Bulej,
Tomáš Bureš,
Vojtěch Horký,
Jaroslav Keznikl

In proceedings of the **4th ACM/SPEC International Conference on Performance Engineering (ICPE '13)**.

Published by ACM,
pages 343-346,
ISBN 978-1-4503-1636-1,
April 2013.

The original version is available electronically from the publisher's site at http://dx.doi.org/10.1145/2479871.2479922.

Summary of the Paper

The focus of this paper, which was published as [BBHK13], is application of the architecture abstractions of DEECo in relevant problem domains, namely in the domain of adaptive deployment in mobile and ad-hoc cloud systems [GKSS11, KDMF10].

The main contribution of this paper is that it illustrates how DEECo abstractions help at the architecture level efficiently address the RDS challenges of dynamism, communication-link instability, and open-endedness, which are equally manifested in the open and dynamic character of the ad-hoc cloud (i.e., challenges C1-C3 outlined in Section 1.3). This way, the paper supports and evaluates the contributions of this thesis w.r.t. the research goals **G1** and (partially) **G3**.

In particular, the paper presents a vision of a DEECo-based software architecture that is able to efficiently handle adaptive planning of application-component deployment and migration in ad-hoc clouds. The vision of the paper is motivated by the following scenario. A user travelling in a bus is working on a mobile device (e.g., a tablet). Her device registers the presence of an offload server machine located in the bus itself, and to save battery, it offloads the application components featuring computationally intensive tasks to that machine. When the bus approaches its destination, the offload server notifies the mobile device that its service will soon become unavailable and components will start migrating back to the device. At the destination, the mobile device will potentially discover another offload server, provided for example by a bus terminal authority, and the offloading process will repeat.

The DEECo-based architecture presented in the paper is based on the idea of exploiting the stateless, best-effort, and dynamic nature of ensemble knowledge exchange to overcome the dynamic availability of the offload nodes, which is caused mainly by the mobility of the device primarily responsible for the application being offloaded/migrated.

Specifically, ensemble knowledge exchange is employed to continuously advertise the offloading requests to the eligible offload nodes (as specified declaratively via ensemble membership; e.g., nodes within 2 network hops). Each request includes information about an application component to be offloaded. The relevant offload nodes (i.e., both the eligible ones and the one where the application component is currently deployed) then set up monitoring components (a.k.a. monitors) to assess the (potential) performance of the application component in that particular deployment (e.g., response time, energy consumption, etc.). This phase differs between the node that actually runs the application component and the other offload nodes (i.e., the former one can use direct measurements, while the latter ones need to employ some prediction methods). Using another ensemble, this information is continuously being advertised back to the mobile device primarily responsible for the application, which uses the information to decide the optimal deployment. If the deployment decision leads to a migration, all the involved offload nodes are notified (via an ensemble) to adjust their monitors. Note that the architecture focuses solely on the task of planning and assumes an external mechanism for the actual migration.

In a way, the architecture (i.e., the advertisement of offload requests, monitoring, advertisement of monitoring results, and planning) forms a closed feedback loop that continuously evaluates the optimal deployment. This way, the declarative nature of ensemble membership enables the architecture to reconfigure freely based on the dynamic availability of the offload nodes.

As an aside, in a similar vein we have elaborated several other applications of DEECo, mostly in cooperation with Volkswagen AG. This includes design and implementation of variants of the cooperative vehicle navigation scenario [SRA+11] featured by the ASCENS project; the results, however, are not publicly available as they fall under a non-disclosure agreement. An excerpt of these results can be found in [SHP+13, SMP+12]. Another application of DEECo is overviewed in [BGAA14]. Finally, in [MKH+13] we have put the ideas presented in this paper into the context of voluntary cloud computing researched within the ASCENS project.

Comments on Authorship

Although the idea of adaptive deployment in ad-hoc clouds based on performance estimations is of equal authorship, I significantly contributed to its elaboration in terms of a DEECo-based architecture. In addition, with the help of the other authors, I authored a majority of the text.

Adaptive Deployment in Ad-Hoc Systems Using Emergent Component Ensembles: Vision Paper

Lubomír Bulej[1,2] Tomáš Bureš[1,2] Vojtěch Horký[1] Jaroslav Keznikl[1,2]

[1]Charles University in Prague
Faculty of Mathematics and Physics
Malostranské náměstí 25
118 00 Prague 1, Czech Republic

[2]Academy of Sciences of the Czech Republic
Institute of Computer Science
Pod Vodárenskou věží 2
182 07 Prague 8, Czech Republic

{bulej,bures,horky,keznikl}@d3s.mff.cuni.cz

ABSTRACT

Mobile cloud computing in the context of ad-hoc clouds brings new challenges when offloading computation from mobile devices. The management of application deployment needs to ensure that the offloading provides users with the expected benefits, but it suddenly needs to cope with a highly dynamic environment which lacks a central authority and in which computational nodes appear and disappear.

We propose an approach to the management of ad-hoc systems in such dynamic environment using component ensembles that connect mobile devices with more powerful computation nodes. Our approach aims to address the challenges of scalability and robustness of such systems without the need for central authority, relying instead on simple patterns that lead to reasonable adaptation decisions based on limited and imprecise information.

Categories and Subject Descriptors

[**Computer systems organization**]: Other architectures — Self-organizing autonomic computing, Distributed architectures — Cloud computing; [**Software and its engineering**]: Extra-functional properties — Software performance

Keywords

ad-hoc cloud, ensembles, adaptive deployment

1. INTRODUCTION

Increasing capabilities of handheld devices and improvements in mobile network infrastructures pave the way for mobile cloud computing [1], an architectural solution where mobile devices offload computation to the cloud to gain advantage for example in increased computing power or reduced battery usage. Another motivation is the emergence of ad-hoc clouds [2], whose computing power comes from pooled resources of nearby general-purpose computing devices rather than from dedicated servers. Our work, carried out in the scope of the ASCENS project [3], aims to combine and extend the two trends by blurring the traditionally strict [4] boundary between the client devices and the cloud infrastructure. We envision an ad-hoc cloud formed as a multitude of dynamically emerging groups of computational devices that share their computing power. The groups will typically involve a number of mobile devices along with locally situated general-purpose computers, potentially connected to a remotely situated dedicated cloud infrastructure.

Important specifics of our ad-hoc cloud concept are that it has a dynamic, mostly uncontrollable architecture with fluctuating computing power and partially limited resources, e.g. the available battery charge for mobile devices. The promise of the ad-hoc cloud is in maintaining the usual benefits associated with offloading computation to a cheap, flexible and resilient environment—however, the ad-hoc cloud applications must dynamically adapt their deployment to deliver the expected user experience in such specific conditions.

The challenge in application adaptation is related to the open character of the ad-hoc cloud. Where a common cloud application can react to increased utilization by requesting additional computing resources, an application in an ad-hoc cloud must act in presence of other adapting applications that share the same resources. Even the scheduling solutions for computational grids, which do cope with shared resources, assume a degree of centralization knowledge and control over the grid that is not available in the ad-hoc cloud [5]. We therefore believe that the application of adaptation solutions in ad-hoc clouds will not assume the shape of complex algorithms that compute close-to-optimal deployment under dynamic conditions, but rather the shape of relatively simple patterns that lead to reasonable adaptation decisions relying on limited and imprecise information.

In this vision paper, we focus on the problem of adaptive deployment planning in ad-hoc clouds and outline an adaptation approach based on a component system with emergent component ensembles [6, 3]. We assume an external mechanism would be responsible for the actual migration [7]. We discuss the potential benefits and scalability of this approach.

2. MOTIVATING EXAMPLE

Besides smart phones, we consider tablet computers to be a perfect target for mobile computing in ad-hoc clouds—modern applications take advantage of their relatively high computational power and users tend to use them both for work and entertainment. However, they are still constrained by the limited battery life.

We start our vision with an example, where we consider a user travelling in a train or a bus, who wants to do productive work using her tablet computer or review travel plans and accommodation. Her tablet registers the presence of an offload server machine located in the bus itself, and to save battery, it offloads most computationally intensive tasks to that machine. Later, when the bus approaches the destination, the offload server notifies her tablet that its service will soon become unavailable and tasks will start moving back to the tablet. When the bus enters the terminal, the tablet will discover another offload server, provided by the terminal authority, and move some of its tasks to the newly found machine.

Many similar examples can be found, and they would follow similar pattern. Abstracting away from the details, we can try to capture such examples formally under one general umbrella. Assume a mobile device M (tablet in our example) and two stationary devices S and T (offload servers in our example). M executes application A, which is internally split into two parts: a frontend Af, responsible for the interaction with a user, and a backend Ab, responsible for the computationally intensive tasks.

In our scenario, M discovers S and assesses that offloading the computationally intensive Ab to S could save M's battery. After some time, S signals that it is going to be unavailable, but M discovers that there are other devices available. Ab is thus migrated to the one that appears most suitable for running Ab—device T in our scenario.

The challenge is in predicting which deployment scenario will—in the context of ad-hoc cloud—deliver the expected user experience. We assume that each application will have a simple performance model that, given specifics of the execution environment and other constraints, will provide a rough estimate of the expected user experience (e.g. what frame rate could be achieved given CPU and GPU budget). The application deployment would be then planned dynamically, taking into account the expected user experience estimated by the model, possibly corrected for measured accuracy of the model from past deployments.

3. COMPONENT ENSEMBLES

Although the scenario of the running example is relatively straightforward, it is relatively difficult to realize due to the inherent dynamicity of the whole ecosystem (i.e. all applications and devices). Further, the combination of the dynamicity and the autonomy of the applications and devices imply the absence of the notion of a global state. In fact, every information about the ecosystem has the form of a "belief" – i.e. an information valid to only a certain extent.

To cope with these issues, we suggest in this paper to take advantage of a component system based on emergent component ensembles [6, 3] and use it to represent situation in such a dynamic ecosystem and to manage the belief about it. To this end, we outline in the rest of the section the basic principles of component ensembles and explain their use in addressing the adaptation in Section 4.

Emergent component ensembles are based on the idea of implicit communication via implicit bindings. Specifically, an ensemble is a dynamically formed group of components, where a component constitutes knowledge (i.e., data) and processes (i.e., active threads operating upon the knowledge). The membership of a component in an ensemble is determined dynamically (the task of the component system runtime framework) according to the membership condition of the ensemble specified upon the knowledge of the components. In an ensemble one component plays the role of the ensemble's coordinator while others play the role of members. A single component can be member and/or coordinator of multiple ensembles at the same time; thus an ensemble forms an independent logical overlay over components.

The interaction among the components forming an ensemble takes the form of knowledge exchange, carried out implicitly (by the runtime framework); i.e., the runtime framework transfers knowledge from one component of the ensemble to another independently on the components' execution.

The benefit of such ensembles is that they allow for capturing communication (i.e., exchange of knowledge) among a (potentially) large, declaratively defined set of components in a concise way.

4. ADAPTATION ARCHITECTURE

To address the aforementioned challenges, we present a generic architecture of adaptation logic, based on emergent component ensembles. The proposed *adaptation architecture* follows the following basic principles.

To ensure separation of concerns, the adaptation logic forms a separate overlay architecture mirroring the architecture of the adapted application.

Additionally, the entities important for deciding adaptation, i.e., (a) computation nodes (and their NFPs), (b) individual adapted applications (and their NFP preferences), and (c) the applications' components (and their NFPs), are explicitly reflected in the adaptation architecture.

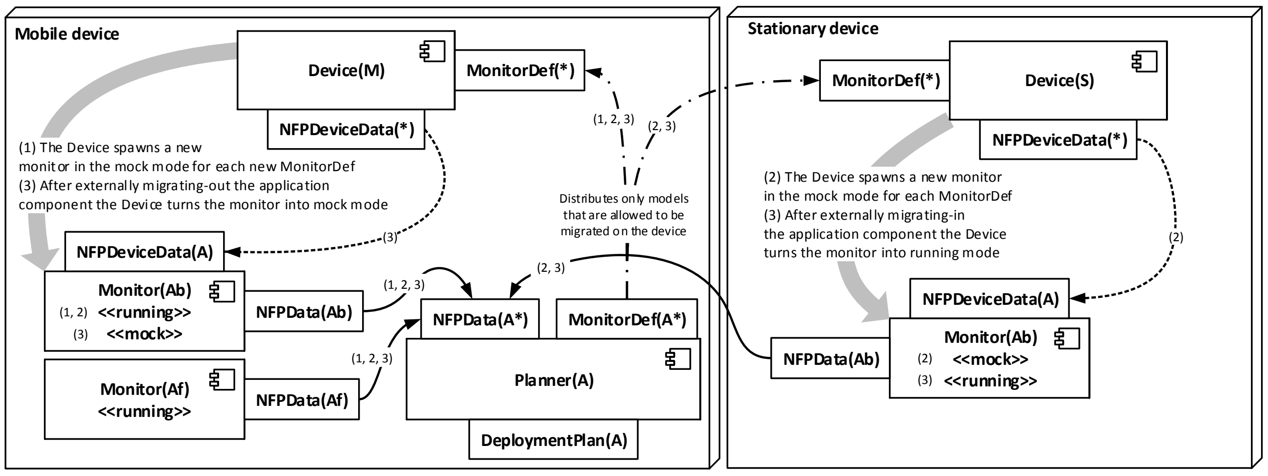
4.1 Adaptation architecture components

The adaptation architecture (Figure 1) is formed by following components:

Planner. Each adapted application, and particularly its NFP preferences, are represented by the Planner component. Specifically, the Planner selects a (potentially optimal) deployment of the application, given the alternatives for deploying each of the application's components. We assume an external mechanism [7] to interpret the deployment plan provided by the Planner and perform the adaptation (e.g., by migrating a component). The alternatives comprise important NFP-related data (NFPData) indicating the (potential) performance of the corresponding application component in that particular deployment (e.g., FPS, energy, etc.). The Planner also advertises definitions of Monitors for individual application components (MonitorDef); see Device.

Monitor. Each application component, particularly each of its deployment alternatives, is reflected by the Monitor component, which is responsible for obtaining the NFPData for that particular alternative. Monitor operates in one of the two modes, depending on the actual deployment of the corresponding application component.

- Monitor is in the *running mode* if it resides on the same computation node as the corresponding application component, i.e. it reflects the actual deployment. NFPData is obtained by performance measurement and analysis of the running application component.
- Monitor is in the *mock mode* if it resides on a different computation node, i.e. it represents a potential deployment alternative. NFPData is obtained from



Mobile device
Device(M)
MonitorDef(*)
NFPDeviceData(*)
(1) The Device spawns a new monitor in the mock mode for each new MonitorDef
(3) After externally migrating-out the application component the Device turns the monitor into mock mode
NFPDeviceData(A)
(3)
Monitor(Ab)
(1, 2) <<running>>
(3) <<mock>>
NFPData(Ab)
Monitor(Af) <<running>>
NFPData(Af)
(1, 2, 3)
(1, 2, 3)
NFPData(A*)
MonitorDef(A*)
Planner(A)
DeploymentPlan(A)
(1, 2, 3)
(2, 3)
(2, 3)
Distributes only models that are allowed to be migrated on the device
Stationary device
MonitorDef(*)
Device(S)
NFPDeviceData(*)
(2) The Device spawns a new monitor in the mock mode for each MonitorDef
(3) After externally migrating-in the application component the Device turns the monitor into running mode
(2)
NFPDeviceData(A)
NFPData(Ab)
Monitor(Ab)
(2) <<mock>>
(3) <<running>>

Figure 1: Adaptation architecture of the running example: phases 1 (M isolated), 2 (S discovered), and 3 (Ab migrated to S). Phases 1,2,3 are in the figure denoted by (1), (2), (3).

the included performance dependency model of the corresponding application component (e.g., the function $CPU \times GPU \rightarrow FPS$). In other words, Monitor predicts – based on the model – the performance of the application component if it would be deployed on that computation node. The model might depend on particular machine-specific performance data (NFPDeviceData, e.g., available CPU speed, etc.); see Device.

Device. Each computation node is reflected by the Device component. Specifically, a Device component ensures management of the Monitors (e.g., it instantiates Monitors advertised by newly discovered Planners) and it provides NFPDeviceData for Monitors in the mock mode.

4.2 Adaptation architecture ensembles

The expectation is that the number of available computation nodes, as well as the number of Monitors, changes dynamically. Therefore, the communication among the components exploits the concept of emergent component ensembles. The architecture involves the following ensembles (Figure 1):

Planner and Device(s). Each Planner is a coordinator of an ensemble that distributes MonitorDefs (including the performance dependency model) of application components to Devices representing currently available computation nodes (including the one the Planner is running on). The Planner is able to constraint which MonitorDefs should be distributed to which Devices (effectively constraining the potential migration destinations for a particular application component). A simplified example of a definition of this ensemble is in Figure 2. It specifies that only reachable devices within 2 network hops are to be considered and that this check is to be performed every 15 seconds. The distribution of the MonitorDefs is performed by adding the MonitorDef to the target component's knowledge.

Planner and Monitor(s). Each Planner is a coordinator of an ensemble that aggregates NFPData from all Monitors corresponding to the components of the application reflected by the Planner. Thus, this ensemble aggregates all the deployment alternatives for the application.

```
ensemble PlannerToDevice:
  coordinator: Planner
  member: Device
  membership: HopDistance(Planner.device, Device) ≤ 2
  knowledge exchange:
    Device.monitorDef[Planner.app] := Planner.monitorDef
  scheduling: periodic( 15s )
```

Figure 2: Example of an ensemble definition.

Device and Monitor(s). Each Device component is a coordinator of an ensemble that distributes NFPDeviceData to the Monitors in the mock mode residing on the corresponding computation node.

4.3 Adaptation architecture in action

In this section, we illustrate on the motivation example the adaptation architecture interaction at runtime.

At first (phase 1, Figure 1), the ensemble distributes the MonitorDefs of both Af and Ab from Planner of A to the Device component of the mobile device (M), which subsequently spawns Monitors for both components and sets them to the running mode. The Monitors start measuring NFPData of the running components which are then aggregated back to the Planner. So far no deployment alternatives are discovered.

After the stationary device (S) is discovered (phase 2, Figure 1), the ensemble propagates MonitorDefs of the components that could be (potentially) migrated (i.e., Ab) to its Device component, which spawns a new Monitor. Since Ab is deployed on a different Device this Monitor runs in the mock mode. Thus, the Device component of the stationary device feeds the Monitor with NFPDeviceData allocated for A. Based on this NFPDeviceData and the performance dependency model of Ab the Monitor produces NFPData reflecting the expected performance of Ab on S. Consequently, another ensemble aggregates all the currently produced NFPData for Af and Ab to the Planner. The Planner thus eventually discovers that there are two deployment alternatives for Ab (i.e., one actually running on M and one modeled on S) and finally decides to deploy Ab on the stationary device.

After Ab is migrated to the stationary device (phase 3, Figure 1), the Monitor on S is set to the running mode, while the Monitor on M is set to the mock mode and the whole monitoring and planning process repeats.

In the case of discovering further stationary devices, new Monitors in the mock mode are spawned which eventually results in new deployment alternatives aggregated in the Planner (similarly, if devices disappear).

5. BENEFITS

Scalability and robustness. By exploiting the features of the ensembles, the adaptation architecture scales well with the number of computation nodes, applications, and components per application. In fact, the adaptation architecture does not require any changes when increasing the number of nodes/applications/components. Furthermore, it is very robust with respect to emergence of computation nodes.

Transparent trade-off management. Due to the declarative nature of ensembles, it is possible to easily manage the trade-offs between the benefit of migration and the effort necessary for monitoring and planning. For instance, Monitors do not have to be spawned on all available nodes but only on a subset; e.g., only the nodes in the same subnet.

Respecting interests of all involved parties without central authority. Although each application is planned autonomously, it is possible (without any centralized authority) to take into account the interests of the other applications and of host devices by regulation of the NFPDeviceData and management of the application's Monitors — e.g., the NFPDeviceData may reflect only a portion of device's resources.

Flexible NFP data acquisition. The NFPData produced by Monitors may contain any information important for deciding adaptation as along as it is obtainable via measurements and/or performance dependency model, e.g., latency between Ab and Af, expected up-time of the computation node (for detecting shutdowns), etc. Moreover, a Monitor can decide between accepting NFPDeviceData given by Device and measuring its own, e.g., Monitor(Ab) can either individually measure latency to Af or rely on the network latency information given by Device(S). Although the performance dependency model employed by a Monitor will usually provide only a rough approximation of the expected performance, it can be potentially improved by actual measurements.

Scalable extensions. Being declarative, the ensembles allow the design to scale with respect to potential extensions of the basic architecture. For instance the Planner itself can be subject to migration in case the application does not have any frontend. Additionally, when understanding the Planner as an entity controlling the NFPs of the application, it is possible to foresee the existence of multiple Planners per application, thus hierarchically decomposing the adaptation.

6. RELATED WORK

In our previous work [8] we proposed to use Stochastic Performance Logic (SPL) [9] to express rules for adaptation in component systems based on real and predicted performance of individual components. The rules controlling the adaptation are similar to the decision logic of the Planner that compares deployment alternatives for Ab.

The issue and challenges of dynamic deployment adaptation has been formulated [4] and addressed [10] previously. However, in spite of the variety of solutions to the individual challenges (e.g., parameters of decision, migration to stationary only or also to mobile devices, acquisition of NFP-related data, etc.), in the majority of the approaches a predetermined solution is used. On the other hand, our adaptation architecture is dynamic enough to allow for combining multiple solutions simultaneously and selecting among them dynamically. It also provides general means to address the remaining challenges (e.g., scheduling of NFP data acquisition or estimation of cost for running before real execution).

A significant body of work has been devised in the related area of Mobile Cloud Computing (MCC) [1]. Although many of the challenges and solutions can be adopted in ad-hoc clouds, there is a significant difference in perceiving the role of the mobile device, i.e., MCC considers the mobile device as separate from the cloud while ad-hoc clouds do not distinguish among the role of the devices.

We assume an external mechanism responsible for the deployment/migration-related aspects of our approach since it has been intensively researched separately, e.g., in [7].

7. CONCLUSION

In this paper, we have presented our vision on addressing the problem of planning deployment adaptation in ad-hoc clouds. In particular, we have described a generic architecture for deployment adaptation logic that is based on the concept of emergent component ensembles. We have also discussed the potential benefits and scalability of this architecture.

8. ACKNOWLEDGEMENTS

This work has been supported by EU project 257414 ASCENS and GACR project P202/10/J042 FERDINAND.

9. REFERENCES

[1] L. Guan *et al.*, "A survey of research on mobile cloud computing," in *Proc. ICIS'11*, pp. 387–392, IEEE CS, 2011.

[2] G. N. C. Kirby *et al.*, "An approach to ad hoc cloud computing," *CoRR*, vol. abs/1002.4738, 2010.

[3] M. Hölzl *et al.*, "Engineering Ensembles: A White Paper of the ASCENS Project." ASCENS Deliverable JD1.1, 2011. Online: `http://www.ascens-ist.eu`.

[4] B.-G. Chun and P. Maniatis, "Dynamically partitioning applications between weak devices and clouds," in *Proc. MCS'10*, pp. 1–5, ACM, 2010.

[5] C. Jiang *et al.*, "A survey of job scheduling in grids," in *Advances in Data and Web Management*, vol. 4505 of *LNCS*, pp. 419–427, Springer, 2007.

[6] J. Keznikl *et al.*, "Towards Dependable Emergent Ensembles of Components: The DEECo Component Model," in *Proc. WICSA/ECSA'12*, IEEE, 2012.

[7] S.-H. Hung *et al.*, "Executing mobile applications on the cloud: Framework and issues," *Computers & Mathematics with Applications*, vol. 63, no. 2, 2012.

[8] L. Bulej *et al.*, "Performance awareness in component systems: Vision paper," in *Proc. COMPSAC'12 Workshops*, pp. 514–519, IEEE CS, 2012.

[9] L. Bulej *et al.*, "Capturing Performance Assumptions using Stochastic Performance Logic," in *Proc. ICPE'12*, pp. 311–322, ACM, 2012.

[10] M. Sharifi, S. Kafaie, and O. Kashefi, "A survey and taxonomy of cyber foraging of mobile devices," *IEEE Commun. Surveys Tuts.*, vol. 14, 2012.

Chapter 4

Conclusion & Open Challenges

This thesis has introduced and elaborated the *DEECo component model*, which encapsulates the architecture abstractions tailored for building dynamic software architectures of *Resilient Distributed Systems* (RDS). The thesis has also introduced the corresponding methods for design, implementation, and analysis of DEECo-based architectures. In this respect, the research goals **G1**-**G3** outlined in Section 1.4 have been achieved. The contributions of this thesis have been presented in terms of a commented collection of co-authored publications, listed in Chapter 3. Specifically, the thesis has presented the DEECo architecture abstractions, centered on the component ensemble concept, which allow for building open-ended, dynamic software architectures that are resilient to communication-link instability and frequent changes in the observable application context, as is often the case in RDS (**G1**). It has also introduced the Invariant Refinement Method (IRM) – a formally grounded design method that embraces the specifics of these architecture abstractions and enables dependable design of DEECo-based architectures (**G2**). In addition, the thesis has presented ARCAS – a formally grounded method for specification and automated synthesis of software connectors that enables scalable, open-ended design and realization of component bindings in face of architecture heterogeneity and dynamism, typical for RDS (**G2**). Further, the thesis has detailed the semantics of the DEECo abstractions w.r.t. distributed and decentralized execution, including a mapping into Java and an execution environment prototype – jDEECo (**G3**). It has also elaborated and evaluated this semantics in the context of formal verification via model checking (**G3**). Finally, the thesis has illustrated the benefits of DEECo by providing a vision of a dynamic architecture for an RDS ensuring adaptive task deployment in ad-hoc cloud systems (**G1, G3**).

In summary, since software architecture design for RDS is a very broad and challenging area that has gained attention only recently, this thesis does not attempt to address every aspect of the challenges related to dynamic architectures of RDS, such as efficient middleware-level implementation. Instead, it aims at clarifying the crucial aspects in order to provide a potential baseline for further research in this area.

To conclude this thesis, the remainder of this chapter presents the author's subjective vision of the open research challenges related to the area of dynamic software architectures for RDS:

Emergent behavior vs. dependability. One of the important challenges of RDS and similar domains, such as cyber-physical systems, is the clash between dependability and emergent behavior (i.e., unforeseen behavior resulting from interactions of many elements of a system or from interaction with an unpredictable environment). While dependability, usually governed by a predictable and analyzable software design, depends on limiting or eliminating emergent behavior, open-endedness and autonomy typically require facilitating the emergent behavior. Although this thesis has contributed to solving this issue at the level of software architecture abstractions (e.g., component ensembles follow strict and predictable design-time prescriptions, while still being established according to the unforeseen situations emerging at runtime), the discrepancy between dependability and emergent behavior remains a very important open challenge, especially in the context of software-architecture design and analysis.

One of the possible research directions in this context is based on the idea of meta-adaptation, i.e., adaptation of adaptation mechanisms themselves via employing models@runtime [PMC+12]. Specifically, while the meta-adaptation can be designed to employ strictly predictable and dependable adaptation mechanisms, it still enables reacting flexibly enough to cope with the emergent situations at runtime. An interesting step towards this direction in RDS that are displaying emergent behavior due to an unpredictable environment is to design the meta-adaptation in such a way that the dependability degrades gradually in a controlled manner. Our initial attempt of pursuing this direction, accompanied with an extension of jDEECo with support for models@runtime, is presented in [BGH+14b].

Provided that the emergent behavior can be approximated stochastically, another promising direction is integration of quantitative verification techniques [Kwi07], which enable formal verification of systems that exhibit stochastic behavior, into software architecture design. This can even include application of quantitative verification for planning adaptation at runtime [CGKM12].

Fighting inaccuracy. Components in decentralized, distributed systems akin to RDS maintain belief about the state of the other components and/or of the real state of their (physical) environment. Because of the distribution and periodic nature of real state sensing, a belief is necessarily outdated (stale). This implies inherent belief inaccuracy (i.e., a deviation of the belief from the real state), which negatively affects the correctness and safety attributes of RDS. The problem lies in the fact that inaccuracy is often hidden in an RDS architecture behind the assumption that components operate correctly in "normal" cases when their belief is not "too stale". Hence, one of the important open challenges is to explicitly acknowledge and address the issue of belief inaccuracy during architecture design.

To this end, an important property of belief in RDS is that it often reflects a real state that changes gradually (e.g., position, battery capacity, temperature). Consequently, one can take advantage of the physical laws that govern the evolution of such real state to estimate/predict the inaccuracy of the belief. Therefore, a promising idea is to establish explicit safety bounds on such predicted belief inaccuracy at the level of architecture in order to capture the margins of safe component operation. In [AABG+14b], we have discussed how such safety bounds can be employed for runtime architecture adaptation.

Exploiting specifics of RDS. The previous open challenges have primarily aimed at solving particular challenges that make software-architecture design for RDS difficult. However, it would be wrong to perceive all the specifics of RDS as obstacles impeding software development, since these specifics also provide new opportunities for getting around the software-engineering challenges in RDS. In this perspective, it is desirable to take advantage of such RDS specifics instead of aiming at adaptation of traditional approaches (e.g., adopting an RDS-specific programming model instead of building a complex middleware to provide a traditional programming model).

One of the promising specifics is the physical mobility. Mobile devices can carry information while moving, which contributes to the overall connectedness of the system by bringing the information across otherwise disconnected network partitions. For example, a vehicle moving along a street segment can aggregate temperature data measured by independent sensors located in the tarmac along its route (which themselves cannot reach any external network) and publish the data on a remote server.

Another interesting aspect of RDS that can be potentially exploited is the locality of information, meaning that the utility of certain measurable system attributes depends on the physical location where these attributes were measured. This has the potential to contribute to system robustness (e.g., sensor-data sharing among nearby devices in face of sensor failures) and scalability (e.g., information does not need to be shared beyond its “area of effect”). Our initial attempt of investigating this opportunity, supported by an experimental, fully decentralized implementation of jDEECo targeting mobile ad-hoc networks, is presented in [BGH+14a].

This is definitely not a complete list of such RDS specifics. Therefore, searching for further specifics that could be advantageously exploited for designing RDS is, by itself, one of the most interesting open challenges.

References

[AABG+13] R. Al Ali, T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil. DEECo computational model – I. Technical Report D3S-TR-2013-01, Dep. of Distributed and Dependable Systems, Charles University in Prague, February 2013. Available online: http://d3s.mff.cuni.cz/publications/.

[AABG+14a] R. Al Ali, T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil. DEECo: an Ecosystem for Cyber-Physical Systems. In *ICSE '14: Companion Proceedings of the 36th International Conference on Software Engineering*, pages 610–611. ACM, June 2014. Poster and extended abstract. Available online: http://-d3s.mff.cuni.cz/publications/.

[AABG+14b] R. Al Ali, T. Bures, I. Gerostathopoulos, J. Keznikl, and F. Plasil. Architecture Adaptation Based on Belief Inaccuracy Estimation. In *WICSA '14: Proceedings of the 11th Working IEEE/IFIP Conference on Software Architecture*. IEEE CS, April 2014.

[ABZ12] D. B. Abeywickrama, N. Bicocchi, and F. Zambonelli. SOTA: Towards a general model for self-adaptive systems. In *WETICE '12: Proceedings of the 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 48–53. IEEE, 2012.

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[ADLMW09] J. Andersson, R. De Lemos, S. Malek, and D. Weyns. Reflecting on self-adaptive software systems. In *SEAMS '09: Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 38–47. IEEE, 2009.

[AG97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):213–249, 1997.

[ASCN03] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language support for connector abstractions. In *ECOOP '03: Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 74–102. Springer, 2003.

[AZ12] D. B. Abeywickrama and F. Zambonelli. Model checking goal-oriented requirements for self-adaptive systems. In *ECBS '12: Proceedings of the 19th International Conference and Workshops on Engineering of Computer Based Systems*, pages 33–42. IEEE, 2012.

[AZI09] D. Athanasopoulos, A. V. Zarras, and V. Issarny. Service substitution revisited. In *ASE'09: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 555–559. IEEE, 2009.

[BB12] K. Beetz and W. Bohm. Challenges in Engineering for Software-Intensive Embedded Systems. In *Model-Based Engineering of Embedded Systems*, pages 3–14. Springer, 2012.

[BBB+11] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software*, 28(3), 2011.

[BBB+12] A. Basu, S. Bensalem, M. Bozga, B. Delahaye, and A. Legay. Statistical abstraction and model-checking of large heterogeneous systems. *International Journal on Software Tools for Technology Transfer*, 14(1):53–72, 2012.

[BBB+13] J. Barnat, N. Benes, T. Bures, I. Cerna, J. Keznikl, and F. Plasil. Towards Verification of Ensemble-Based Component Systems. In *FACS '13: Proceedings of the 10th International Symposium on Formal Aspects of Component Software*, volume 8348 of *Lecture Notes in Computer Science*. Springer, October 2013. In press. Available online: http://d3s.mff.cuni.cz/publications/.

[BBC+06] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, et al. SCC: A Service Centered Calculus. In *Web services and formal methods*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer Berlin Heidelberg, 2006.

[BBCO12] N. Benes, B. Buhnova, I. Cerna, and R. Oslejsek. Reliability Analysis in Component-based Development via Probabilistic Model Checking. In *CBSE '12: Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*, pages 83–92. ACM, 2012.

[BBCP13] J. Barnat, N. Benes, I. Cerna, and Z. Petruchova. DCCL: verification of component systems with ensembles. In *CBSE '13: Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, pages 43–52. ACM, 2013.

[BBF09] G. Blair, N. Bencomo, and R. B. France. Models@run.time. *Computer*, 42(10):22–27, 2009.

[BBG+06] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling. Symbolic invariant verification for systems with dynamic structural adaptation. In *ICSE '06: Proceedings of the 28th International Conference on Software engineering*, pages 72–81. ACM, 2006.

[BBH+12] L. Bulej, T. Bures, V. Horky, J. Keznikl, and P. Tuma. Performance Awareness in Component Systems: Vision Paper. In *COMPSACW '12: Proceedings of the 36th IEEE Annual Computer Software and Applications Conference Workshops*, pages 514–519. IEEE Computer Society, July 2012.

[BBH+13] J. Barnat, L. Brim, V. Havel, J. Havlicek, J. Kriho, M. Lenco, P. Rockai, V. Still, and J. Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *CAV '13: Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 863–868. Springer, 2013.

[BBHK13] L. Bulej, T. Bures, V. Horky, and J. Keznikl. Adaptive Deployment in Ad-Hoc Systems Using Emergent Component Ensembles: Vision Paper. In *ICPE '13: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 343–346. ACM, April 2013.

[BBK+12] L. Bulej, T. Bures, J. Keznikl, A. Koubkova, A. Podzimek, and P. Tuma. Capturing Performance Assumptions Using Stochastic Performance Logic. In *ICPE '12: Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 311–322. ACM, April 2012.

[BBK14] M. Babka, T. Balyo, and J. Keznikl. Solving SMT Problems with a Costly Decision Procedure by Finding Minimum Satisfying Assignments of Boolean Formulas. In R. Lee, editor, *Software Engineering Research, Management and Applications*, volume 496 of *Studies in Computational Intelligence*, pages 231–246. Springer International Publishing, 2014.

[BBNS09] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis. D-Finder: A tool for compositional deadlock detection and verification. In *CAV '09: Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 614–619. Springer, 2009.

[BBS06] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12. IEEE, 2006.

[BC11] E. Borde and J. Carlson. Towards verified synthesis of ProCom, a component model for real-time embedded systems. In *CBSE '11: Proceedings of the 14th international ACM Sigsoft Symposium on Component Based Software Engineering*, pages 129–138. ACM, 2011.

[BCC+05] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[BCC+09] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. In A. Biere, M. J. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 457–481. IOS Press, 2009.

[BCD+09] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Perez. GCM: a grid extension to Fractal for autonomous distributed components. *Annals of telecommunications*, 64(1-2):5–24, 2009.

[BCGZ06] G. Brown, B. H. C. Cheng, H. Goldsby, and J. Zhang. Goal-oriented Specification of Adaptation Requirements Engineering in Adaptive Systems. In *SEAMS '06: Proceedings of the 2006 International Workshop on Self-adaptation and Self-managing Systems*, pages 23–29. ACM, 2006.

[BCL+06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal component model and its support in Java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[BDFR08] D. Bertrand, A.-M. Deplanche, S. Faucou, and O. Roux. A Study of the AADL Mode Change Protocol. In *ICECCS '08: Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems*, pages 288–293, 2008.

[BFVW06] R. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying Multi-agent Programs by Model Checking. *Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, 2006.

[BGAA14] T. Bures, I. Gerostathopoulos, and R. Al Ali. DEECo: Software Engineering for Smart CPS. *ERCIM news Special theme: Cyber-Physical Systems*, (97), April 2014. Available online: http://ercim-news.ercim.eu/en97/special/deeco-software-engineering-for-smart-cps.

[BGF+08] N. Bencomo, P. Grace, C. Flores, D. Hughes, and G. Blair. Genie: Supporting the model driven development of reflective, component-based adaptive systems. In *ICSE' 08: Proceedings of the 30th International Conference on Software engineering*, pages 811–814. ACM, 2008.

[BGH+13] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil. DEECo: an Ensemble-Based Component System. In *CBSE '13: Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, pages 81–90. ACM, June 2013.

[BGH+14a] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil. Gossiping Components for Cyber-Physical Systems. In *ECSA '14: Proceedings of the 8th European Conference on Software Architecture*. Springer, August 2014. Accepted for publication. Available online: http://d3s.mff.cuni.cz/publications/.

[BGH+14b] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, F. Plasil, and N. Plouzeau. Adaptation in Cyber-Physical Systems: from System Goals to Architecture Configurations. Technical Report D3S-TR-2014-01, Dep. of Distributed and Dependable Systems, Charles University in Prague, January 2014. Available online: http://d3s.mff.cuni.cz/publications/.

[BGT05] S. Burmester, H. Giese, and M. Tichy. Model-driven development of reconfigurable mechatronic systems with mechatronic UML. In *Model Driven Architecture*, volume 3599 of *Lecture Notes in Computer Science*, pages 47–61. Springer Berlin Heidelberg, 2005.

[BHH+06] H. Baumeister, F. Hacklinger, R. Hennicker, A. Knapp, and M. Wirsing. A component model for architectural programming. *Electronic Notes in Theoretical Computer Science*, 160:75–96, 2006.

[BHM09] T. Bures, P. Hnetynka, and M. Malohlava. Using a product line for creating component systems. In *SAC '09: Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 501–508. ACM, 2009.

[BHP06] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SEAA '06: Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48. IEEE, 2006.

[BHP09] E. Borde, G. Haik, and L. Pautet. Mode-based reconfiguration of critical software component architectures. In *DATE'09: Design, Automation & Test in Europe Conference & Exhibition*, pages 1160–1165. IEEE, 2009.

[BHR14] F. Baude, L. Henrio, and C. Ruz. Programming distributed and adaptable autonomous components—the GCM/ProActive framework. *Software: Practice and Experience*, 2014. Available online: http://dx.doi.org/10.1002/spe.2270.

[BHvMW09] A. Biere, M. J. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[BJC05] T. Batista, A. Joolia, and G. Coulson. Managing dynamic reconfiguration in component-based systems. In *Software Architecture*, volume 3527 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2005.

[BJM+11] T. Bures, P. Jezek, M. Malohlava, T. Poch, and O. Sery. Strengthening Component Architectures by Modeling Fine-grained Entities. In *SEAA '11: Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 124–128. IEEE, 2011.

[BJMS12] M. Bozga, M. Jaber, N. Maris, and J. Sifakis. Modeling Dynamic Architectures Using Dy-BIP. In T. Gschwind, F. Paoli, V. Gruhn, and M. Book, editors, *Software Composition*, volume 7306 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012.

[BKR09] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[BL06] M. Bichier and K.-J. Lin. Service-Oriented Computing. *Computer*, 39(3):99–101, 2006.

[BLP01] L. Bettini, M. Loreti, and R. Pugliese. Modelling node connectivity in dynamically evolving networks. *Electronic Notes in Theoretical Computer Science*, 54:81–91, 2001.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *ACM SIGPLAN Notices*, volume 36, pages 203–213. ACM, 2001.

[Box98] D. Box. *Essential COM*. Addison-Wesley Professional, 1998.

[BP04] T. Bures and F. Plasil. Communication Style Driven Connector Configurations. In *SERA '03: Software Engineering Research and Applications*, volume 3026 of *Lecture Notes in Computer Science*, pages 102–116. Springer, 2004.

[BPG+04] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[BPG13] J. M. Barnes, A. Pandey, and D. Garlan. Automated planning for software architecture evolution. In *ASE '13: Proceedings of the 2013 IEEE/ACM 28th International Conference on Automated Software Engineering*, pages 213–223. IEEE, 2013.

[BPR01] F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with JADE. In *Intelligent Agents VII Agent Theories Architectures and Languages*, volume 1986 of *Lecture Notes in Computer Science*, pages 89–103. Springer, 2001.

[BRHL99] P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents-Components for Intelligent Agents in Java. *AgentLink News Letter*, 2(1):2–5, 1999.

[BS08] S. Bliudze and J. Sifakis. The algebra of connectors—structuring interaction in BIP. *IEEE Transactions on Computers*, 57(10):1315–1330, 2008.

[BSG+09] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Muller, M. Pezze, and M. Shaw. Engineering Self-Adaptive Systems through Feedback Loops. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 48–70. Springer, 2009.

[Bur06] T. Bures. *Generating connectors for homogeneous and heterogeneous deployment*. PhD thesis, Charles University in Prague, 2006.

[But11] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 3rd edition, 2011.

[BWR09] A. Barker, C. D. Walton, and D. Robertson. Choreographing web services. *IEEE Transactions on Services Computing*, 2(2):152–166, 2009.

[CAC08] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. *ACM Trans. Softw. Eng. Methodol.*, 17(2):7:1–7:52, May 2008.

[CBG+08] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Transactions on Computer Systems (TOCS)*, 26(1):1, 2008.

[CCL06] I. Crnkovic, M. Chaudron, and S. Larsson. Component-based development process and component lifecycle. In *ICSEA '06: Proceedings of the International Conference on Software Engineering Advances*. IEEE, 2006.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. Technical report, W3C, 2001. Available online: http://www.w3.org/TR/wsdl.

[CCP11] J. Cubo, C. Canal, and E. Pimentel. Context-Aware Composition and Adaptation based on Model Transformation. *Journal of Universal Computer Science*, 17(5):777–806, 2011.

[CDLG+09] B. H. Cheng, R. De Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software engineering for self-adaptive systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009.

[CDT13] A. Cimatti, M. Dorigatti, and S. Tonetta. OCRA: A tool for checking the refinement of temporal contracts. In *ASE '13: Proceedings of the 28th International Conference on Automated Software Engineering*, pages 702–705. IEEE, April 2013.

[CFMTS10] J. Carlson, J. Feljan, J. Maki-Turja, and M. Sjodin. Deployment modelling and synthesis in a component model for distributed embedded systems. In *SEAA '10: Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 74–82. IEEE, 2010.

[CG98] L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 1998.

[CGKM12] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive Software Needs Quantitative Verification at Runtime. *Commun. ACM*, 55(9):69–77, September 2012.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT press, Cambridge, MA, USA, 1999.

[CH04] H. Cervantes and R. S. Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 614–623. IEEE Computer Society, 2004.

[CL02] I. Crnkovic and M. P. H. Larsson. *Building reliable component-based software systems*. Artech House Publishers, Norwood, MA, USA, 2002.

[CL10] F. Calzolai and M. Loreti. Simulation and Analysis of Distributed Systems in Klaim. In *Coordination Models and Languages*, volume 6116 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 2010.

[CN02] P. Clements and L. Northrop. *Software product lines: practices and patterns*. Addison-Wesley, 2002.

[CNW01] F. Curbera, W. Nagy, and S. Weerawarana. Web services: Why and how. In *Proceedings of the OOPSLA 2001 Workshop on Object-Oriented Web Services*. ACM, 2001.

[CT12] A. Cimatti and S. Tonetta. A property-based proof system for contract-based design. In *SEAA'12: Proceedings of the 38th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 21–28. IEEE, 2012.

[CVZ+11] I. Crnkovic, A. Vulgarakis, M. Zagar, A. Petricic, J. Feljan, L. Lednicki, and J. Maras. Classification and Survey of Component Models. In *SoftCOM 2011: DICES Workshop at the 19th International Conference on Software, Telecommunications and Computer Networks*, 2011.

[DFB+12] E. Daubert, F. Fouquet, O. Barais, G. Nain, G. Sunye, J.-M. Jezequel, J.-L. Pazat, and B. Morin. A models@ runtime framework for designing and managing service-based applications. In *2012 Workshop on European Software Services and Systems Research-Results and Challenges (S-Cube)*, pages 10–11. IEEE, 2012.

[DHDG06] C. Dubois, T. Hardin, and V. Donzeau-Gouge. Building certified components within FOCAL. *Trends in Functional Programming*, 5:33–48, 2006.

[dJ09] M. de Jonge. Developing product lines with third-party components. *Electronic Notes in Theoretical Computer Science*, 238(5):63–80, 2009.

[DL06] P.-C. David and T. Ledoux. An aspect-oriented approach for developing self-adaptive Fractal components. In *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2006.

[DLGM+13] R. De Lemos, H. Giese, H. A. Muller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, et al. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 1–32. Springer, 2013.

[DMSFR10] G. Di Marzo Serugendo, J. Fitzgerald, and A. Romanovsky. MetaSelf: an architecture and a development method for dependable self-* systems. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 457–461. ACM, 2010.

[DNFLP13] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese. A Language-Based Approach to Autonomic Computing. In B. Beckert, F. Damiani, F. Boer, and M. Bonsangue, editors, *FMCO 2011*, volume 7542 of *Lecture Notes in Computer Science*, pages 25–48. Springer Berlin Heidelberg, 2013.

[DNFP98] R. De Nicola, G. L. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[DNGM+08] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3-4):313–341, 2008.

[DNL04] R. De Nicola and M. Loreti. A modal logic for mobile agents. *ACM Transactions on Computational Logic (TOCL)*, 5(1):79–128, 2004.

[DNLPT14] R. De Nicola, M. Loreti, R. Pugliese, and F. Tiezzi. A formal approach to autonomic systems programming: the SCEL Language. *ACM Transactions on Autonomous and Adaptive Systems*, 2014. In press. Available online: http://-eprints.imtlucca.it/2117/.

[DR14] A. K. Dwivedi and S. K. Rath. Analysis of a Complex Architectural Style C2 Using Modeling Language Alloy. *Computer Science and Information Technology*, 2(3):152–164, 2014.

[DVL96] R. Darimont and A. Van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 179–190. ACM, 1996.

[EHH+13] T. Eckardt, C. Heinzemann, S. Henkler, M. Hirsch, C. Priesterjahn, and W. Schafer. Modeling and verifying dynamic communication structures based on graph transformations. *Computer Science-Research and Development*, 28(1):3–22, 2013.

[EHL07] C. Escoffier, R. S. Hall, and P. Lalanda. iPOJO: An extensible service-oriented component framework. In *Proceedings of the 2013 IEEE International Conference on Services Computing*, pages 474–481. IEEE, 2007.

[FGH06] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical report, DTIC Document, 2006. Available online: http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA455842.

[FGM04] J. Ferber, O. Gutknecht, and F. Michel. From Agents to Organizations: An Organizational View of Multi-agent Systems. In P. Giorgini, J. Müller, and J. Odell, editors, *Agent-Oriented Software Engineering IV*, volume 2935 of *Lecture Notes in Computer Science*, pages 214–230. Springer Berlin Heidelberg, 2004.

[FGR⁺07] R. Friedman, D. Gavidia, L. Rodrigues, A. C. Viana, and S. Voulgaris. Gossiping on MANETs: The Beauty and the Beast. *SIGOPS Oper. Syst. Rev.*, 41(5):67–74, October 2007.

[FHS⁺06] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, 2006.

[FL10] J. L. Fiadeiro and A. Lopes. A model for dynamic reconfiguration in service-oriented architectures. In M. Babar and I. Gorton, editors, *Software Architecture*, volume 6285 of *Lecture Notes in Computer Science*, pages 70–85. Springer, 2010.

[FMF⁺12] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jezequel. A dynamic component model for cyber physical systems. In *CBSE '12: Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*, pages 135–144. ACM, 2012.

[FPMT01] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specifications in Tropos. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, pages 174–181. IEEE, 2001.

[Fre10] A. Freeman. Windows Presentation Foundation. In *Introducing Visual C# 2010*, pages 1069–1098. Springer, 2010.

[FUMK03] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *ASE '03: Proceedings of 18th IEEE International Conference on Automated Software Engineering*, pages 152–161. IEEE, 2003.

[GBH13] I. Gerostathopoulos, T. Bures, and P. Hnetynka. Position Paper: Towards a requirements-driven design of ensemble-based component systems. In *Proceedings of the 2013 International Workshop on Hot topics in Cloud Services*, pages 79–86. ACM, 2013.

[GCH⁺04] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[GCW⁺02] T. Genssler, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arevalo, B. Schonhage, P. Muller, and C. Stich. Components for embedded software: the PECOS approach. In *CASES '02: Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 19–26. ACM, 2002.

[GKB+14] I. Gerostathopoulos, J. Keznikl, T. Bures, M. Kit, and F. Plasil. Software Engineering for Software-Intensive Cyber-Physical Systems. Technical Report D3S-TR-2014-02, Dep. of Distributed and Dependable Systems, Charles University in Prague, January 2014. Available online: http://d3s.mff.cuni.cz/publications/.

[GKSS11] L. Guan, X. Ke, M. Song, and J. Song. A survey of research on mobile cloud computing. In *Proceedings of the 2011 10th IEEE/ACIS International Conference on Computer and Information Science*, pages 387–392. IEEE Computer Society, 2011.

[GLMS11] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: a toolbox for the construction and analysis of distributed processes. In *TACAS '11: Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer, 2011.

[GLPT12] E. Gjondrekaj, M. Loreti, R. Pugliese, and F. Tiezzi. Modeling adaptation with a tuple-based coordination language. In *SAC '12: Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1522–1527. ACM, 2012.

[GMK02] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *Proceedings of the first workshop on Self-healing systems*, pages 33–38. ACM, 2002.

[GMW00] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In *Foundations of component-based systems*, pages 47–68. Cambridge University Press, 2000.

[GNT04] M. Ghallab, D. Nau, and P. Traverso. *Automated planning: theory & practice*. Elsevier, 2004.

[GPC04] D. Giannakopoulou, C. S. Pasareanu, and J. M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 211–220. IEEE Computer Society, 2004.

[GS03] J. Greenfield and K. Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 16–27. ACM, 2003.

[GS07] D. Garlan and B. Schmerl. Architecture-driven modelling and analysis. In *Proceedings of the eleventh Australian workshop on Safety critical systems and software-Volume 69*, pages 3–17. Australian Computer Society, Inc., 2007.

[HB13] C. Heinzemann and S. Becker. Executing reconfigurations in hierarchical component architectures. In *CBSE '13: Proceedings of 16th International ACM SIGSOFT Symposium on Component Based Software Engineering*, pages 3–12. ACM, 2013.

[HC01] G. T. Heineman and W. T. Councill, editors. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Her10] S. Herold. Checking architectural compliance in component-based systems. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2244–2251. ACM, 2010.

[HI10] K. M. Hansen and M. Ingstrup. Modeling and analyzing architectural change with Alloy. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2257–2264. ACM, 2010.

[HK10] L. Henrio and M. U. Khan. Asynchronous Components with Futures: Semantics and Proofs in Isabelle/HOL. *Electronic Notes in Theoretical Computer Science*, 264(1):35–53, 2010.

[HK14] R. Hennicker and A. Klarl. Foundations for Ensemble Modeling–The Helena Approach. In *Specification, Algebra, and Software*, volume 8373 of *Lecture Notes in Computer Science*, pages 359–381. Springer Berlin Heidelberg, 2014.

[HKMU06] D. Hirsch, J. Kramer, J. Magee, and S. Uchitel. Modes for Software Architectures. In V. Gruhn and F. Oquendo, editors, *Software Architecture*, volume 4344 of *Lecture Notes in Computer Science*, pages 113–126. Springer Berlin Heidelberg, 2006.

[HKR09] L. Henrio, F. Kammüller, and M. Rivera. An Asynchronous Distributed Component Model and Its Semantics. In F. Boer, M. Bonsangue, and E. Madelaine, editors, *Formal Methods for Components and Objects*, volume 5751 of *Lecture Notes in Computer Science*, pages 159–179. Springer Berlin Heidelberg, 2009.

[HM08] M. C. Huebscher and J. A. McCann. A Survey of Autonomic Computing - Degrees, Models, and Applications. *ACM Comput. Surv.*, 40(3):7:1–7:28, August 2008.

[HMM11] P. Hnetynka, L. Murphy, and J. Murphy. Comparing the Service Component Architecture and Fractal Component Model. *Computer Journal*, 54(7), 2011.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[HP06] P. Hnetynka and F. Plasil. Dynamic reconfiguration and access to services in hierarchical component models. In *Component-Based Software Engineering*, volume 4063 of *Lecture Notes in Computer Science*, pages 352–359. Springer, 2006.

[HPB$^+$10] P. Hosek, T. Pop, T. Bures, P. Hnetynka, and M. Malohlava. Comparison of Component Frameworks for Real-Time Embedded Systems. In L. Grunske, R. Reussner, and F. Plasil, editors, *Component-Based Software Engineering*, volume 6092 of *Lecture Notes in Computer Science*, pages 21–36. Springer Berlin Heidelberg, 2010.

[HPMS11] R. Hall, K. Pauls, S. McCulloch, and D. Savage. *OSGi in action: Creating modular applications in Java*. Manning Publications Co., 2011.

[HRW08] M. Holzl, A. Rauschmayer, and M. Wirsing. Engineering of Software-Intensive Systems: State of the Art and Research Challenges. In *Software-Intensive Systems and New Computing Paradigms*, volume 5380 of *Lecture Notes in Computer Science*, pages 1–44. Springer, 2008.

[HS05] M. N. Huhns and M. P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[HZPK07] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. Rapid Prototyping of Distributed Real-Time Embedded Systems Using the AADL and Ocarina. In *IEEE International Workshop on Rapid System Prototyping*, pages 106–112. IEEE, 2007.

[HZPK08] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite. *ACM Trans. Embed. Comput. Syst.*, 7(4), August 2008.

[IBB11] V. Issarny, A. Bennaceur, and Y.-D. Bromberg. Middleware-Layer Connector Synthesis: Beyond State of the Art in Middleware Interoperability. In *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 217–255. Springer, 2011.

[IFMW08] F. Irmert, T. Fischer, and K. Meyer-Wegener. Runtime adaptation in a service-oriented component model. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 97–104. ACM, 2008.

[ISJ+09] V. Issarny, B. Steffen, B. Jonsson, G. Blair, P. Grace, M. Kwiatkowska, R. Calinescu, P. Inverardi, M. Tivoli, A. Bertolino, et al. CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 154–161. IEEE, 2009.

[IST11] P. Inverardi, R. Spalazzese, and M. Tivoli. Application-layer connector synthesis. In *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 148–190. Springer, 2011.

[Jac02] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[Jac12] D. Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[JEA+07] D. Jordan, J. Evdemon, A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, et al. Web services business process execution language version 2.0. *OASIS standard*, 2007. Available online: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[JPK12] P. Jancik, P. Parizek, and J. Kofron. BeJC: Checking Compliance Between Java Implementation and Behavior Specification. In *WCOP '12: Proceedings of the 17th International Doctoral Symposium on Components and Architecture*, pages 31–36. ACM, 2012.

[JS00] D. Jackson and K. Sullivan. COM revisited: tool-assisted modelling of an architectural framework. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 149–158. ACM, 2000.

[KBP+13] J. Keznikl, T. Bures, F. Plasil, I. Gerostathopoulos, P. Hnetynka, and N. Hoch. Design of Ensemble-Based Component Systems by Invariant Refinement. In *CBSE '13: Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, pages 91–100. ACM, June 2013.

[KBPH14] J. Keznikl, T. Bures, F. Plasil, and P. Hnetynka. Automated resolution of connector architectures using constraint solving (ARCAS method). *Software & Systems Modeling*, 13(2):843–872, May 2014.

[KBPK12] J. Keznikl, T. Bures, F. Plasil, and M. Kit. Towards Dependable Emergent Ensembles of Components: The DEECo Component Model. In *WICSA/ECSA '12: Proceedings of the Joint 10th Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture*, pages 249–252. IEEE Computer Society, August 2012.

[KC03] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[KDMF10] G. N. C. Kirby, A. Dearle, A. Macdonald, and A. A. A. Fernandes. An Approach to Ad-hoc Cloud Computing. *CoRR*, abs/1002.4738, 2010.

[KG10] J. S. Kim and D. Garlan. Analyzing architectural styles. *Journal of Systems and Software*, 83(7):1216–1235, 2010.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97: Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[KM07] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE'07: Proceedings of the 2007 Workshop on the Future of Software Engineering*, pages 259–268. IEEE, 2007.

[KM09] J. Kramer and J. Magee. A Rigorous Architectural Approach to Adaptive Software Engineering. *Journal of Computer Science and Technology*, 24(2):183–188, 2009.

[KMBH11] J. Keznikl, M. Malohlava, T. Bures, and P. Hnetynka. Extensible Polyglot Programming Support in Existing Component Frameworks. In *SEAA '11: Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 107–115. IEEE Computer Society, August 2011.

[KRKH09] J. E. Kim, O. Rogalla, S. Kramer, and A. Hamann. Extracting, specifying and predicting software system properties in component based real-time embedded software development. In *Proceedings of the 31st International Conference on Software Engineering - Companion Volume*, pages 28–38. IEEE, 2009.

[Kwi07] M. Kwiatkowska. Quantitative Verification: Models Techniques and Tools. In *ESEC-FSE '07: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 449–458. ACM, 2007.

[LBP+08] D. Le Berre, A. Parrain, et al. On SAT technologies for dependency management and beyond. In *ASPL'08: First Workshop on Software Product Lines*, pages 197–200, 2008.

[LCS13] L. Lednicki, J. Carlson, and K. Sandstrom. Model Level Worst-case Execution Time Analysis for IEC 61499. In *CBSE '13: Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, pages 169–178. ACM, 2013.

[LLC10] M. Leger, T. Ledoux, and T. Coupaye. Reliable Dynamic Reconfigurations in a Reflective Component Model. In L. Grunske, R. Reussner, and F. Plasil, editors, *Component-Based Software Engineering*, volume 6092 of *Lecture Notes in Computer Science*, pages 74–92. Springer Berlin Heidelberg, 2010.

[LN11] J. W. Lloyd and K. S. Ng. Declarative programming for agent applications. *Autonomous Agents and Multi-Agent Systems*, 23(2):224–272, 2011.

[LPH04] H. Liu, M. Parashar, and S. Hariri. A component-based programming model for autonomic applications. In *Proceedings of the International Conference on Autonomic Computing*, pages 10–17. IEEE, 2004.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[LQR09] A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A Model Checker for the Verification of Multi-Agent Systems. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 682–688. Springer Berlin Heidelberg, 2009.

[LVL02] E. Letier and A. Van Lamsweerde. Deriving operational software specifications from system goals. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 119–128. ACM, 2002.

[LW07] K.-K. Lau and Z. Wang. Software Component Models. *IEEE Transactions on Software Engineering*, 33(10):709–724, 2007.

[Mal12] M. Malohlava. *Variability of Execution Environments for Component-based Systems*. PhD thesis, Charles University in Prague, 2012.

[MBB$^+$12] J.-E. Mehus, T. Batista, J. Buisson, et al. ACME vs PDDL: support for dynamic reconfiguration of software architectures. In *CAL'12: Proceedings of Conference Francophone sur les Architectures Logicielles*, pages 48–57, 2012. Available online: http://hal.archives-ouvertes.fr/hal-00703176/.

[MBDC$^+$06] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE'06. Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 199–208. IEEE, 2006.

[MBJ$^+$09] B. Morin, O. Barais, J. Jezequel, F. Fleurey, and A. Solberg. Models at runtime to support dynamic adaptation. *Computer*, 42(10):44–51, 2009.

[MBNJ09] B. Morin, O. Barais, G. Nain, and J.-M. Jezequel. Taming dynamically adaptive systems using models and aspects. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 122–132. IEEE Computer Society, 2009.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Software Engineering—ESEC'95*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153. Springer, 1995.

[MK96] J. Magee and J. Kramer. Dynamic structure in software architectures. *ACM SIGSOFT Software Engineering Notes*, 21(6):3–14, 1996.

[MK99] J. Magee and J. Kramer. *Concurrency: State models and java programs*. John Wiley & Sons, Inc., 1999.

[MKG99] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In P. Donohoe, editor, *Software Architecture*, volume 12 of *The International Federation for Information Processing*, pages 35–49. Springer, 1999.

[MKH+13] P. Mayer, A. Klarl, R. Hennicker, M. Puviani, F. Tiezzi, R. Pugliese, J. Keznikl, and T. Bures. The Autonomic Cloud: A Vision of Voluntary, Peer-2-Peer Cloud Computing. In *SASOW '13: Proceedings of the IEEE 7th International Conference on Self-Adaptation and Self-Organizing Systems Workshops*, pages 89 – 94. IEEE Computer Society, September 2013.

[MMP00] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, pages 178–187. ACM, 2000.

[MO06] R. Mateescu and F. Oquendo. π-AAL: an architecture analysis language for formally specifying and verifying structural and behavioural properties of software architectures. *ACM SIGSOFT Software Engineering Notes*, 31(2):1–19, 2006.

[MPBH13] M. Malohlava, F. Plasil, T. Bures, and P. Hnetynka. Interoperable domain-specific languages families for code generation. *Software: Practice and Experience*, 43(5):479–499, 2013.

[MPP08] M. Morandini, L. Penserini, and A. Perini. Towards goal-oriented development of self-adaptive systems. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 9–16. ACM, 2008.

[MPS08] H. Muller, M. Pezze, and M. Shaw. Visibility of control in adaptive systems. In *Second International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008), ICSE 2008 Workshop*, 2008.

[MPT13] A. Margheri, R. Pugliese, and F. Tiezzi. Linguistic Abstractions for Programming and Policing Autonomic Computing Systems. In *UIC/ATC '13: Proceedings of the 10th International Conference on Ubiquitous Intelligence and Computing and 10th International Conference on Autonomic and Trusted Computing*, pages 404–409, Dec 2013.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and computation*, 100(1):1–40, 1992.

[MR09] J. Marino and M. Rowley. *Understanding SCA (Service Component Architecture)*. Pearson Education, 2009.

[MS+08] P. Merle, J.-B. Stefani, et al. A formal specification of the Fractal component model in Alloy. Technical Report RR-6721, INRIA, 2008. Available online: hal.inria.fr/-inria-00338987.

[NTER06] J. Nakazawa, H. Tokuda, W. K. Edwards, and U. Ramachandran. A Bridging Framework for Universal Interoperability in Pervasive Systems. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*. IEEE, 2006.

[NWP02] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, 2002.

[OHJ+99] P. Oreizy, D. Heimbigner, G. Johnson, M. M. Gorlick, R. N. Taylor, A. L. Wolf, N. Medvidovic, D. S. Rosenblum, and A. Quilici. An architecture-based approach to self-adaptive software. *IEEE Intelligent systems*, 14(3):54–62, 1999.

[Oqu04] F. Oquendo. π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–14, 2004.

[Pap03] M. P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering*, pages 3–12. IEEE, 2003.

[PBVDL05] K. Pohl, G. Bockle, and F. Van Der Linden. *Software product line engineering*. Springer Berlin Heidelberg, 2005.

[PCHW12] T. Patikirikorala, A. Colman, J. Han, and L. Wang. A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *SEAMS '12: Proceedings of the 2012 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 33–42. IEEE, 2012.

[PHH+13] T. Pop, P. Hnetynka, P. Hosek, M. Malohlava, and T. Bures. Comparison of component frameworks for real-time embedded systems. *Knowledge and Information Systems*, 2013. In press. Available online: http://dx.doi.org/10.1007/s10115-013-0627-9.

[PKH+11] T. Pop, J. Keznikl, P. Hosek, M. Malohlava, T. Bures, and P. Hnetynka. Introducing support for embedded and real-time devices into existing hierarchical component system: Lessons learned. In *SERA '11: Proceedings of the 9th International Conference on Software Engineering Research, Management and Applications*, pages 3–11. IEEE CS, August 2011.

[PMC+12] G. Perrouin, B. Morin, F. Chauvel, F. Fleurey, J. Klein, Y. Le Traon, O. Barais, and J.-M. Jezequel. Towards Flexible Evolution of Dynamically Adaptive Systems. In *ICSE '12: Proceedings of the 34th International Conference on Software Engineering*, pages 1353–1356. IEEE Press, 2012.

[PMM+07] C. Ponsard, P. Massonet, J. F. Molderez, A. Rifaut, A. van Lamsweerde, and H. Van Tran. Early verification and validation of mission critical systems. *Formal Methods in System Design*, 30(3):233–247, 2007.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–144. Springer-Verlag New York, Inc., 1985.

[POS06] J. Polakovic, A. E. Ozcan, and J.-B. Stefani. Building reconfigurable component-based OS with THINK. In *SEAA '06: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 178–185. IEEE, 2006.

[PP10] P. Parizek and F. Plasil. Assume-guarantee verification of software components in sofa 2 framework. *IET Software*, 4:210–221, June 2010.

[PPK06] P. Parizek, F. Plasil, and J. Kofron. Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker. In *SEW '06: Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, pages 133–141. IEEE Computer Society, 2006.

[PPO+12] T. Pop, F. Plasil, M. Outly, M. Malohlava, and T. Bures. Property Networks Allowing Oracle-based Mode-change Propagation in Hierarchical Components. In *CBSE '12: Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*, pages 93–102. ACM, 2012.

[PS08] C. Peper and D. Schneider. Component engineering for adaptive ad-hoc systems. In *SEAMS '08: Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, pages 49–56. ACM, 2008.

[PSPK13] T. Poch, O. Sery, F. Plasil, and J. Kofron. Threaded behavior protocols. *Formal Aspects of Computing*, 25(4):543–572, 2013.

[PWT+08] M. Prochazka, R. Ward, P. Tuma, P. Hnetynka, and J. Adamek. A component-oriented framework for spacecraft on-board software. In *DASIA'08: Proceedings of Data Systems In Aerospace*, volume 665 of *ESA Special Publication*. European Space Agency, 2008.

[RBAF10] N. Rehman, Ur, S. Bibi, S. Asghar, and S. Fong. Comparative Study of Goal-Oriented Requirements Engineering. In *NISS '10: Proceedings of the 4th International Conference on New Trends in Information Science and Service Science*, pages 248–253. IEEE, 2010.

[RBF+08] J. F. Rolland, J. P. Bodeveix, M. Filali, D. Chemouil, and D. Thomas. Modes in Asynchronous Systems. In *ICECCS '08: Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems*, pages 282–287. IEEE Computer Society, 2008.

[RC04] J. Real and A. Crespo. Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[RCGT09] A. Radermacher, A. Cuccuru, S. Gerard, and F. Terrier. Generating execution infrastructures for component-oriented specifications with a model driven toolchain: a case study for MARTE's GCM and real-time annotations. In *ACM SIGPLAN Notices*, volume 45, pages 127–136. ACM, 2009.

[RG+95] A. S. Rao, M. P. Georgeff, et al. BDI Agents: From Theory to Practice. In *ICMAS' 95: In Proceedings of the First International Conference on Multi-Agent Systems*, volume 95, pages 312–319. AAAI Press, 1995.

[RLSS10] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-Physical Systems: The Next Computing Revolution. In *DAC '10: Proceedings of the 47th Design Automation Conference*, pages 731–736. ACM, 2010.

[RRS+05] S. Robert, A. Radermacher, V. Seignole, S. Gerard, V. Watine, and F. Terrier. Enhancing Interaction Support in the Corba Component Model. In *From Specification to Embedded Systems Application*, volume 184 of *IFIP On-Line Library in Computer Science*, pages 137–146. Springer, 2005.

[SA09] M. Simonot and V. Aponte. A declarative formal approach to dynamic reconfiguration. In *Proceedings of the 1st international workshop on Open component ecosystems*, pages 1–10. ACM, 2009.

[San93] D. Sangiorgi. *Expressing mobility in process algebras: first-order and higher-order paradigms*. PhD thesis, The University of Edinburgh, 1993.

[SBK13] N. Serbedzija, T. Bures, and J. Keznikl. Engineering Autonomous Systems. In *PCI '13: Proceedings of the 17th Panhellenic Conference on Informatics*, pages 128–135. ACM, September 2013.

[Ser10] O. Sery. *Automated Verification of Software*. PhD thesis, Charles University in Prague, 2010.

[SHP+13] N. Serbedzija, N. Hoch, C. Pinciroli, M. Kit, T. Bures, G. V. Monreale, U. Montanari, P. Mayer, and J. Velasco. Integration and Simulation Report for the ASCENS Case Studies. Deliverable D7.3, 2013. Available online: http://www.ascens-ist.eu/deliverables.

[SI10] R. Spalazzese and P. Inverardi. Mediating connector patterns for components interoperability. In *Software Architecture*, volume 6285 of *Lecture Notes in Computer Science*, pages 335–343. Springer, 2010.

[SLB09] Y. Shoham and K. Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2009.

[SMP+12] N. Serbedzija, M. Massink, C. Pinciroli, M. Brambilla, D. Latella, M. Dorigo, M. Birattari, P. Mayer, J. Velasco, N. Hoch, H. P. Bensler, D. Abeywickrama, J. Keznikl, I. Gerostathopoulos, T. Bures, R. De Nicola, and M. Loreti. Ensemble Model Syntheses with Robot, Cloud Computing and e-Mobility. Deliverable D7.2, 2012. Available online: http://www.ascens-ist.eu/deliverables.

[SMR+12] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Software: Practice and Experience*, 42(5):559–583, 2012.

[SRA+11] N. Serbedzija, S. Reiter, M. Ahrens, J. Velasco, C. Pinciroli, N. Hoch, and B. Werther. Requirement Specification and Scenario Description of the ASCENS Case Studies. Deliverable D7.1, 2011. Available online: http://www.ascens-ist.eu/deliverables.

[SVB+08] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, and I. Crnkovic. A Component Model for Control-Intensive Distributed Embedded Systems. In *Component-Based Software Engineering*, volume 5282 of *Lecture Notes in Computer Science*, pages 310–317. Springer, 2008.

[SW04] H. Schuschel and M. Weske. Automated planning in a service-oriented architecture. In *Proceedings of 13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 75–80. IEEE, 2004.

[Szy02] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[TBKC07] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, pages 95–104. ACM, 2007.

[TGEM10] H. Tajalli, J. Garcia, G. Edwards, and N. Medvidovic. PLASMA: A Plan-based Layered Architecture for Software Model-driven Adaptation. In *ASE '10: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 467–476. ACM, 2010.

[TMD10] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, Inc., 2010.

[TMS10] A. Tiberghien, P. Merle, and L. Seinturier. Specifying self-configurable component-based systems with FracToy. In *Abstract State Machines, Alloy, B and Z*, volume 5977 of *Lecture Notes in Computer Science*, pages 91–104. Springer, 2010.

[TPYZ09] S. Tang, X. Peng, Y. Yu, and W. Zhao. Goal-directed modeling of self-adaptive software architecture. In T. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, and R. Ukor, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 29 of *Lecture Notes in Business Information Processing*, pages 313–325. Springer, 2009.

[TSP+04] W. Tsai, W. Song, R. Paul, Z. Cao, and H. Huang. Services-Oriented Dynamic Reconfiguration Framework for Dependable Distributed Computing. In *COMPSAC '04: Proceedings of the 28th Annual International Computer Software and Applications Conference*, pages 554–559. IEEE Computer Society, 2004.

[VHB+03] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[VL01] A. Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, pages 249–262. IEEE, 2001.

[VL03] A. Van Lamsweerde. From system goals to software architecture. In M. Bernardo and P. Inverardi, editors, *Formal Methods for Software Architectures*, volume 2804 of *Lecture Notes in Computer Science*, pages 25–43. Springer, 2003.

[VLDL98] A. Van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering*, 24(11):908–926, 1998.

[VLL00] A. Van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering*, 26(10):978–1005, 2000.

[VLL04] A. Van Lamsweerde and E. Letier. From object orientation to goal orientation: A paradigm shift for requirements engineering. In M. Wirsing, A. Knapp, and S. Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future*, volume 2941 of *Lecture Notes in Computer Science*, pages 325–340. Springer, 2004.

[VOVDLKM00] R. Van Ommering, F. Van Der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.

[VPK05] T. Vergnaud, L. Pautet, and F. Kordon. Using the AADL to describe distributed applications from middleware to software components. In *Reliable Software Technology–Ada-Europe 2005*, volume 3555 of *Lecture Notes in Computer Science*, pages 67–78. Springer, 2005.

[VSC+09] A. Vulgarakis, J. Suryadevara, J. Carlson, C. Seceleanu, and P. Pettersson. Formal semantics of the ProCom real-time component model. In *SEAA' 09: Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications*, pages 478–485. IEEE, 2009.

[VSCS10] A. Vulgarakis, S. Sentilles, J. Carlson, and C. Seceleanu. Integrating behavioral descriptions into a component model for embedded systems. In *SEAA '10: Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 113–118. IEEE, 2010.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331. IEEE Computer Society, 1986.

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more*. Prentice Hall PTR, 2005.

[WFHP02] M. Wooldridge, M. Fisher, M.-P. Huget, and S. Parsons. Model checking multi-agent systems with MABLE. In *AAMAS '02: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2*, pages 952–959. ACM, ACM, 2002.

[WSO01] N. Wang, D. C. Schmidt, and C. O'Ryan. *Overview of the CORBA Component Model*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[YCH12] H. Yin, J. Carlson, and H. Hansson. Towards mode switch handling in component-based multi-mode systems. In *CBSE '12: Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*, pages 183–188. ACM, 2012.

[YH13] H. Yin and H. Hansson. Mode switch timing analysis for component-based multi-mode systems. *Journal of Systems Architecture*, 59(10):1299–1318, 2013.

[YLL+08] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos, and J. C. Leite. From goals to high-variability software design. In A. An, S. Matwin, Z. W. Ras, and D. Slezak, editors, *Foundations of Intelligent Systems*, volume 4994 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.

[YP04] J. Yang and M. P. Papazoglou. Service components for managing the life-cycle of service compositions. *Information Systems*, 29(2):97–125, 2004.

[YQCH13] H. Yin, H. Qin, J. Carlson, and H. Hansson. Mode Switch Handling for the ProCom Component Model. In *CBSE '13: Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, pages 13–22. ACM, 2013.

[YYP13] L. Yang, S. Yang, and L. Plotnick. How the internet of things technology enhances emergency response operations. *Technological Forecasting and Social Change*, 80(9):1854 – 1867, 2013. Planning and Foresight Methodologies in Emergency Preparedness and Management.

[ZC06] J. Zhang and B. H. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 371–380. ACM, 2006.

[ZML10] P. Zhang, H. Muccini, and B. Li. A classification and comparison of model checking software architecture techniques. *Journal of Systems and Software*, 83(5):723–744, 2010.

Web References

[1] Alloy.
http://alloy.mit.edu/

[2] Apache Software Foundation. JavaSpaces Service Specification.
http://river.apache.org/doc/specs/html/js-spec.html

[3] Apache Software Foundation. Maven.
http://maven.apache.org/

[4] Apache Software Foundation. Tuscany.
http://tuscany.apache.org/

[5] ASCENS: Autonomic Service-Component Ensembles.
www.ascens-ist.eu

[6] Computing Research and Education Association of Australasia (CORE). The CORE Conference Ranking.
http://core.edu.au/index.php/categories/conference%20rankings/1

[7] CHESS: Composition with Guarantees for High-integrity Embedded Software Components Assembly.
http://www.chess-project.org/

[8] DEECo.
http://d3s.mff.cuni.cz/projects/components_and_services/deeco/

[9] The Eclipse Foundation. Equinox p2.
http://www.eclipse.org/equinox/p2/

[10] FraSCAti - Open SCA middleware platform.
http://frascati.ow2.org

[11] GIMPLE Model Checker for C/C++ programs (GMC).
http://d3s.mff.cuni.cz/~sery/gmc/

[12] Google. Google Guice.
http://code.google.com/p/google-guice/

[13] jDEECo.
https://github.com/d3scomp/JDEECo

[14] jRESP: Java Runtime Environment for SCEL Programs.
http://jresp.sourceforge.net/

[15] Kevoree.
http://kevoree.org/

[16] NASA. Java Path Finder.
http://babelfish.arc.nasa.gov/trac/jpf/

[17] OASIS Open CSA. Service Component Architecture.
http://oasis-opencsa.org/sca

[18] Object Management Group. CORBA Component Model Specification v4.0.
http://www.omg.org/spec/CCM/

[19] Oracle. Enterprise JavaBeans specification v3.2.
http://jcp.org/aboutJava/communityprocess/final/jsr345/index.html

[20] Oracle. JavaBeans specification.
http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html

[21] OSGi Alliance. OSGi service platform, core specification, release 5.
http://www.osgi.org/Specifications/HomePage

[22] SpringSource. Spring Framework.
http://www.springsource.org/

[23] XSB Prolog.
http://xsb.sourceforge.net/