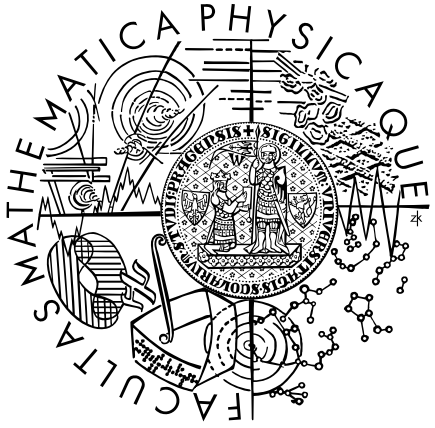
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



Ilias Gerostathopoulos

Model-Driven Development of Software-Intensive Cyber-Physical Systems

Department of Distributed and Dependable Systems

Advisor: Doc. RNDr. Petr Hnětynka, Ph.D.
Study program: Computer Science
Specialization: Software Systems

Prague 2015

Acknowledgement

"As you set out for Ithaka
hope the voyage is a long one,
full of adventure, full of discovery. […]"[1]

And what a wonderful voyage this has been! I would like to express my gratitude to everyone who stood by me to advise, inspire, and encourage me.

First and foremost, I would like to thank my advisors Petr Hnetynka and Tomas Bures for their constant support in all matters of research and administration. Thank you for introducing me to the exciting world of research, for all the interesting conversations we shared (including the non-research-related ones), and for helping me blend in with the department. I would like to express my equal gratitude to Frantisek Plasil for his constant advice, support, and encouragement. I could not have asked for better mentors.

I would also like to thank my friends and colleagues Michal Kit, Rima Al Ali, and Jaroslav Keznikl for being my fellow travelers, for fighting together the *"Laistrygonians and Cyclops"*[1] and for sharing my enthusiasm for research that matters. I would like to thank also my colleagues in the department of Distributed and Dependable Systems who made (each in their own way) this department a wonderful place to work. A particular thank-you goes to Andranik Muradyan, Dominik Skoda, Filip Krijt, Jakub Daniel, Jan Kofron, Jirka Vinarek, Martin Decky, Paolo Arcaini, Pavel Jancik, Pavel Jezek, Pavel Parizek, Peter Libic, Petr Tuma, Viliam Simko, Vladimir Matena, Vojtech Horky, and Zbynek Jiracek. It would be a mistake not to thank also Petra Novotna for her patience and indispensable support in all the administrative tasks.

I am also grateful to the EU project RELATE 264840, part of Marie-Curie Initial Training Network of the 7th Research Framework Programme, that provided financial support throughout my doctoral studies and enabled me to spend two months as a research fellow at INRIA research labs in Rennes, France and two months at SEERC research center in Thessaloniki, Greece.

Above all, I am deeply in debt to my girlfriend Rania, my parents Thanasis and Stella, and the rest of my family for their patience and constant encouragement to *"reach what you cannot"*[2]. This work would not have been completed without you. Σας ευχαριστώ μέσα από την καρδιά μου.

My final thoughts go to my ninety-year-old grandmother, Dimitra (Toula) Gerostathopoulou, who dedicated her life to her husband, sons and grandchildren, being a live example of determination and perseverance. This thesis is dedicated to her.

[1] Constantine P. Cavafy. Ithaka. In *Collected Poems*. Princeton University Press, 1992.

[2] Nikos Kazantzakis. *Report to Greco*. Touchstone, 1975.

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, June 15, 2015 …….……

Ilias Gerostathopoulos

Annotation

Title	*Model-Driven Development of Software-Intensive Cyber-Physical Systems*
Author	Ilias Gerostathopoulos iliasg@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. Petr Hnětynka, Ph.D. hnetynka@d3s.mff.cuni.cz (+420) 221 914 143
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

Software-Intensive Cyber-Physical Systems (siCPS) are modular, open-ended, networked, large-scale embedded Information and Communication Technology (ICT) systems that are increasingly depending on software. They need to be both dependable and flexible to adapt to changes in their dynamic environments. This combination poses challenges in their design and development, as traditional model-driven design and development techniques cannot account for both dependability and self-adaptivity.

The thesis proposes (1) a new, model-based design process for siCPS, which comprises both appropriate methods and models and deals with dependability and self-adaptivity, and (2) a mapping of the design models to implementation-level abstractions, which allows for model-driven development and early experimentations in siCPS.

Specifically, the thesis delivers (1) by introducing and elaborating on the Invariant Refinement Method (IRM), and its extension for self-adaptivity, for the design of siCPS based on the ensemble paradigm. IRM was integrated into the ensemble development life cycle, a methodology for the development of autonomic ensemble-based systems. Contributing to (2), the thesis provides a mapping of the IRM concepts to the concepts of the DEECo components model. The mapping is supported by prototype implementations of model manipulation tools. Finally, the feasibility and effectiveness of the IRM design process has been validated via a controlled experiment.

Keywords

Design process, Software architecture, Self-adaptation

Anotace

Název práce	*Modelem řízený vývoj softwarových cyber-physical systémů*
Autor	Ilias Gerostathopoulos iliasg@d3s.mff.cuni.cz (+420) 221 914 236
Katedra	Katedra distribuovaných a spolehlivých systémů Matematicko-fyzikální fakulta Univerzita Karlova v Praze
Školitel	Doc. RNDr. Petr Hnětynka, Ph.D. hnetynka@d3s.mff.cuni.cz (+420) 221 914 143
Adresa	Katedra distribuovaných a spolehlivých systémů Univerzita Karlova v Praze Malostranské náměstí 25 118 00 Praha
WWW	http://d3s.mff.cuni.cz/

Abstrakt

Softwarové cyber-physical systémy (siCPS) jsou modulární, otevřené, propojené a rozsáhlé ICT systémy, které stále více závisejí na softwaru. Tyto systémy musejí být spolehlivý a zároveň schopné se adaptovat na změny v proměnlivém prostředí, ve kterém jsou provozovány. Tato kombinace činí jejich návrh a realizaci obtížnými, neboť tradiční modelem řízené techniky pro návrh a vývoj nejsou schopny brát v potaz zároveň spolehlivost a autoadaptivitu.

Tato práce navrhuje: (1) nový, modelem řízený proces návrhu siCPS systémů, který obsahuje vhodné metodiky a modely a zároveň podporuje spolehlivost i autoadaptivitu; a (2) mapování vzniklých návrhových modelů do abstrakcí na úrovni implementace, což umožňuje modelem řízený vývoj a rychlé experimentování v kontextu siCPS.

Konkrétní realizaci bodu (1) představuje zavedení a rozpracování metody Invariant Refinement Method (IRM) – a jejího rozšíření pro podporu adaptivity – určené pro návrh siCPS založených na konceptu tzv. ansámblů. IRM je integrováno do metodiky vytvořené pro vývoj autonomních systémů založených na ansámblech. Z hlediska realizace bodu (2) práce navrhuje mapování z IRM konceptů do konceptů komponentového modelu DEECo. Mapování je podloženo implementačními prototypy nástrojů pro manipulace s modelem. Důležitou součástí práce je rovněž kontrolovaný experiment ověřující použitelnost a efektivnost návrhového procesu IRM.

Klíčová slova

Návrhový proces, softwarová architektura, autoadaptivita

Contents

Chapter 1

Introduction

1.1 Towards Software-Intensive Cyber-Physical Systems

The progress in embedded and mobile technologies and networking has brought large potential for building systems that improve human life and address societal, technical and environmental challenges, e.g. energy consumption, ambient assisted living, crisis coordination. In particular, the progress has created the necessary infrastructure to build large-scale pervasive software systems that combine data from various sources to control real-world ecosystems. An example is an emergency coordination system, where first responders' devices communicate seamlessly across organizational boundaries in order to head to the most affected areas in a timely fashion. Another example is an intelligent navigation system, where vehicles communicate with each other and with traffic lights and parking units in order to minimize traffic congestion and optimize parking allocation. A third example is smart exhibition centers, where visitors' devices communicate with stationary cameras and with each other in order to avoid long queues.

These systems fall into the broad category of cyber-physical systems, that is, "engineered systems that are built from, and depend upon, the seamless integration of computational algorithms and physical components" [23]. Cyber-physical systems are considered by the European H2020 research agenda "the next generation of embedded Information and Communication Technology (ICT) systems that are interconnected and collaborating, providing citizens and businesses with a wide range of innovative applications and services" [10].

Compared to traditional embedded systems, cyber-physical systems are modular, dynamic, networked, and large-scale. They are also increasingly depending on software, which has become their most intricate and extensive constituent [HRW08b]. In the context of this thesis, we will call the systems featuring the above properties *Software-Intensive Cyber-Physical Systems* (siCPS).

An important desired feature of siCPS is that they need to be self-adaptive. The self-adaptivity requirement stems from the close connection to the ever-changing physical world. In its general form, self-adaptivity refers to the ability of a software system to change its structure and/or behavior in response to external stimuli and changes in its internal state. Although self-adaptivity, together with individual self-* properties such as self-configuration, self-healing, and self-optimization, has been studied extensively in

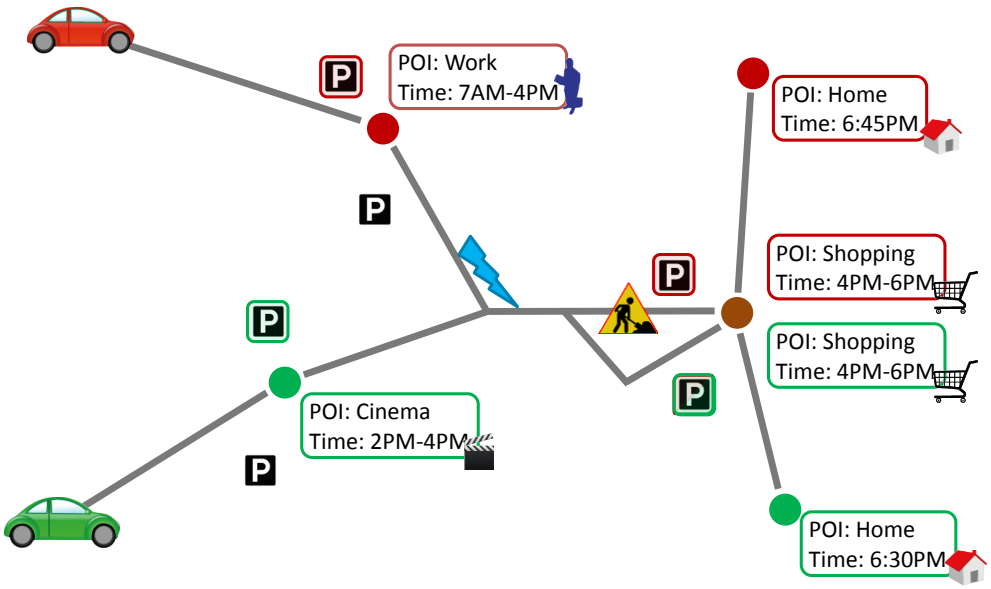


Figure 1. siCPS example: electric mobility case study.

different domains [CLG+09, ST09], the domain of siCPS introduces distinct challenges in the application of existing techniques. These include the need to deal with the unavailability of a global state and the operational uncertainty, two characteristic properties of siCPS [GKB+14].

At the same time, siCPS need to be dependable. The dependability requirement mainly stems from the fact that siCPS often host safety-critical applications. It pertains to safety and predictability concerns at the first place, followed by the continuous availability, privacy and security concerns. As a distinct challenge, siCPS often need to operate in dynamic, often unpredictable and even hostile environments, where the problems of limited network connectivity and physical mobility of devices loom large.

1.1.1 Example of a Software-Intensive Cyber-Physical System

To illustrate the context and challenges of siCPS, the following text describes a scenario based on the electric vehicle mobility case study, a case study of ASCENS FP7 project [SRA+11], which was also used in the evaluation of the thesis (Section 5.1). The scenario consists of drivers, moving around a city in their electric vehicles. Drivers have to reach specific destinations within some time limits, which depend on their daily schedule (Figure 1). Vehicles are equipped with sensors of basic capabilities, e.g. monitoring the battery level of the car, but also more sophisticated ones, e.g. monitoring the traffic level along the route, etc. Vehicles can only park and charge their batteries in designated stations. They can also communicate with each other and with the parking/charging stations that lie within their transmission spectrum. A key challenge is that no central coordination point is assumed; there is no global control or global planning. The whole system can be seen a set of mobile devices which form dynamic communication groups

to serve a specific goal: drivers should arrive to their destinations in time, leveraging the infrastructure resources in a close-to-optimal way. The system operates in a highly dynamic and unpredictable environment, as drivers may change their schedules at runtime, roads may become blocked, sensors may start malfunctioning and emitting inaccurate data, parking and charging stations may get out of communication reach, etc. In order to respond to these changes in the physical environment and the drivers' requirements and to recover from temporary failures, the system has to be self-adaptive, in particular self-healing and self-optimizing. Under these stringent requirements, it also has to be dependable in order to serve the drivers continuously and in a predictable way.

1.2 Problem Statement

Traditional software engineering (SE) of ICT systems relies on a number of assumptions in order to abstract away a lot of complexity pertaining to physical environment and networking via the operating system and middleware stacks. As discussed in [GKB+14], these assumptions include the assumption of static physical structure, stable connections, location obliviousness, reachability and stable connections, availability of global state, focus on reactive behavior, crisp consistency, and controlled dynamism.

Due to the physical distribution at a large scale, the physical mobility, and the dynamicity pertaining to the close connection to the physical environment, the above assumptions do not hold in SE of siCPS. Although some of them may as well be violated when developing ICT systems with special requirements, such as high availability and open-endedness, siCPS stand out by the big number of such assumptions that are violated *at the same time*. This calls for a radical re-thinking of the computing and networking technologies and abstractions in order to provide an adequate foundation for siCPS [Lee08]. It also renders traditional SE methods that rely on the above assumptions non-applicable to siCPS and calls for new SE and in particular architecture design methods that embrace the distinct characteristics of siCPS.

The new SE methods have to tackle the following major challenges:

C1 **Robust context-driven self-adaptivity.** The main challenge in inducing self-adaptivity to siCPS pertains to the inherent dynamism and the unpredictability of the physical environment. The physical substratum continuously evolves as mobile devices move in the environment. As a result, it is hard to obtain a fully up-to-date snapshot of the global system configuration or architecture at runtime, upon which the self-adaptation logic can be based. A robust self-adaptation mechanism for siCPS has thus to deal with partial views of the system and non-fully-up-to-date data.

C2 **Dependability.** This challenge is not specific to siCPS; every software system that hosts critical applications has to be dependable. The extra challenge pertains to controlling the emergent behavior in siCPS, i.e. the behavior that

comes about as the joint product of behaviors and interactions of the different entities of a system. As these entities in siCPS are not purely virtual software entities (software components), but often have their representations in the physical environment, dealing with the multiple interactions and side effects that can arise in the system gets very complex. This challenge manifests itself both at the design and analysis phase, where it pertains to anticipating and designing for the different interactions in the system, and at runtime, where the system has to be monitored for faults and deviations from its original specification.

C3 **Open-endedness and development effort.** Being large-scale pervasive systems, siCPS do not have strict boundaries, i.e. they are inherently open-ended. As classic software development processes, e.g. Object-Oriented Analysis and Design [AN05], rely on delineating the system boundaries upfront in a rather strict way, they have to be adapted for the development of siCPS. At the same time, dealing with large systems imposes that their development has to scale in terms of design, analysis, coding, and testing effort.

C4 **Distributed coordination.** As a siCPS is typically a large distributed system, special attention has to be paid to the means that the different subsystems can coordinate with each other when needed (for example, when subsystems need to collectively apply a system-wise adaptation action). Resorting to a design with one or more arbitrators that would orchestrate the different subsystems is one choice; more decentralized schemes can also be employed. In any case, one has to deal with the fact that strict distributed coordination may become too expensive in siCPS (in terms of messages exchanged, network medium overload, data inconsistency due to temporary disconnections, etc.).

C5 **Operational uncertainty.** An overarching issue in the development of siCPS is the inherent uncertainty related to their infrastructure – *operational* uncertainty. This type of uncertainty arises from the environment in which the system is deployed and, in siCPS, concerns situations such as network unavailability, hardware failures, and unanticipated resource scarcity. This challenge is related to the inherent dynamism of siCPS and their close connection to the continuously changing physical world.

1.3 Research Goals

Responding to the challenges presented in Section 1.2, this thesis focuses on the architecture design phase of SE of dynamic self-adaptive dependable siCPS and aims to provide appropriate design abstractions and processes to address these challenges. The primary intention is to adopt ideas from goal-oriented requirements engineering [VL01],

agent-oriented computing [SLB08], and ensemble-based systems [HRW08a], while focusing on the software engineering aspects.

Since the area is broad, the thesis primarily focuses on clarifying on the appropriate design processes and design models and on the mapping of the latter to implementation-level artifacts. The thesis thus targets the following research goals:

G1 The first goal is to propose an appropriate **design process** for **open-ended** siCPS. The process should be model-based, i.e. rely on appropriate **design models**, to allow automation and early validation. The models and the process should provide both **dependability**, in the form of correct-by-construction guarantees and traceability of low-level design artifacts to system-level goals, and **context-driven self-adaptivity**, in the form of adjusting to different operational contexts and situations.

G2 The second goal is to **map** the proposed design models to **implementation-level abstractions** to allow for model-driven development and early experimentations in siCPS. This goal includes a realization of the mapping in terms of prototypes of model design and manipulation tools that help reduce the **development effort**.

The two goals cut across all the identified challenges, while focusing explicitly on challenges **C1**-**C3**.

1.4 Contribution and Publications

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 [1], and the EU FP7 Marie Curie ITN project RELATE [28] in which the author participated as Early Stage Researcher.

The following peer-reviewed papers and technical reports form the core contribution presented in this thesis. An overview of the contribution is presented in Chapter 3, while the summaries and full texts of these publications are included in Chapter 4.

[GKB+14] I. Gerostathopoulos, J. Keznikl, T. Bures, M. Kit, and F. Plasil. Software Engineering for Software-Intensive Cyber-Physical Systems. In *INFORMATIK 2014: Proceedings of the 44th Annual Meeting of the German Informatics Society*, pages 1179–1190. Gesellschaft für Informatik, Bohn, Germany, September 2014.

[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, April 2013.

[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. Awarded with the ACM Distinguished Paper Award.

[BGK+15] T. Bures, I. Gerostathopoulos, J. Keznikl, F. Plasil, and P. Tuma. Formalization of Invariant Patterns for the Invariant Refinement Method. In R. De Nicola and R. Hennicker, editors, *Software, Services and Systems*, volume 8950 in *Lecture Notes in Computer Science*, pages 602–208. Springer International Publishing, 2015.

[Ger14] I. Gerostathopoulos. Model-Driven Design of Ensemble-Based Component Systems. In *MODELS '14: Proceedings of the ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems Poster Session and the ACM Student Research Competition*, volume 1258, pages 63–68. CEUR-WS.org, September 2014. Awarded with the 2nd place award of the ACM Student Research Competition – Graduate level.

[GBH+15b] I. Gerostathopoulos, T. Bures, P. Hnetynka, J. Keznikl, M. Kit, F. Plasil, and N. Plouzeau. Self-Adaptation in Cyber-Physical Systems: from System Goals to Architecture Configurations. Technical Report D3S-TR-2015-02, Department of Distributed and Dependable Systems, April 2015.

[BDNG+13] T. Bures, R. De Nicola, I. Gerostathopoulos, N. Hoch, M. Kit, N. Koch, G. Valentina Monreale, U. Montanari, R. Pugliese, N. Serbedzija, M. Wirsing, and F. Zambonelli. A Life Cycle for the Development of Autonomic Systems: The e-Mobility Showcase. In *SASOW '13: Proceedings of the 7th IEEE International Conference on Self-Adaptation and Self-Organizing Systems Workshops*, pages 71–76. IEEE, September 2013.

The publications [KBP+13], [BGK+15] and [BDNG+13] are of equal authorship. In [GKB+14], [GBH13], and [GBH+15b] under helpful guidance and supervision of the other authors, I came up with the main idea and authored most of the text. Additionally, in [GBH+15b] I contributed by elaboration and formalization of the main idea, the case study, and the evaluation. Finally, the single authorship of [Ger14] reflects my individual participation in the ACM Student Research Competition at MODELS 2014.

The main contributions of this thesis were also included in the following peer-reviewed poster publication, which is of equal authorship.

[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.

In addition, the following co-authored peer-reviewed publications support the contributions listed above by sharing the underlying topics of software design for siCPS and software self-adaptation.

[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.

[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*, pages 87–90. IEEE, April 2014.

[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*, pages 250–266. Springer, August 2014. Awarded with the Best Research Paper Award.

[AAGGH+14] R. Al Ali, I. Gerostathopoulos, I. Gonzalez-Herrera, A. Juan-Verdejo, M. Kit, and B. Surajbali. An Architecture-Based Approach for Compute-Intensive Pervasive Systems in Dynamic Environments. In *HotTopiCS '14: Proceedings of the 2nd International Workshop on Hot Topics in Cloud service Scalability*. ACM, March 2014. Article no. 3.

[BBG+15] L. Bulej, T. Bures, I. Gerostathopoulos, V. Horky, J. Keznikl, L. Marek, M. Tschaikowski, M. Tribastone, and P. Tuma. Supporting Performance Awareness in Autonomous Ensembles. In M. Wirsing, M. Hölzl, N. Koch, and P. Mayer, editors, *Software Engineering for Collective Autonomic Systems*, volume 8998 in *Lecture Notes in Computer Science*, pages 291–322. Springer International Publishing, 2015.

[BGH+15] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil. The Invariant Refinement Method. In M. Wirsing, M. Hölzl, N. Koch, and P. Mayer, editors, *Software Engineering for Collective Autonomic Systems*, volume 8998 in *Lecture Notes in Computer Science*, pages 405–428. Springer International Publishing, 2015.

[FMA+15] A. Filieri, M. Maggio, K. Angelopoulos, N. D'Ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein, F. Krikava, S. Misailovic, A. V. Papadopoulos, S. Ray, A. M. Sharifloo, S. Shevtsov, M. Ujma, and T. Vogel. Software Engineering Meets Control Theory. In *SEAMS'15: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, May 2015. In press.

[GGK+15] S. Goetz, I. Gerostathopoulos, F. Krikava, A. Shahzada, and R. Spalazzese. Adaptive Exchange of Distributed Partial Models@run.time for Highly Dynamic Systems. In *SEAMS'15: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, May 2015. In press.

[KGB+15] M. Kit, I. Gerostathopoulos, T. Bures, P. Hnetynka, and F. Plasil. An Architecture Framework for Experimentations with Self-Adaptive Cyber-Physical Systems. In *SEAMS'15: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, May 2015. In press.

[GBH+15a] I. Gerostathopoulos, T. Bures, P. Hnetynka, A. Hujecek, F. Plasil, and D. Skoda. Meta-Adaptation Strategies for Adaptation in Cyber-Physical Systems. In *ECSA '15: Proceedings of the 9th European Conference on Software Architecture*. Springer, September 2015. In press.

Notably, the book chapter of [BGH+15] provides a summary of the contributions originally published in [KBP+13] and [GBH+15b].

1.5 Structure

The thesis is structured in the following way. First, Chapter 2 presents the state of the art in requirements modeling and analysis and software development methodologies of large dynamic open-ended systems akin to siCPS, with particular focus on the research goals **G1** and **G2**. In particular, Section 2.1 overviews representative goal-oriented requirements engineering approaches and evaluates their applicability in the domain of siCPS. Similarly, Section 2.2 overviews representative agent-oriented and component-based software engineering approaches and provides a critical evaluation of their applicability in siCPS. Finally, Section 2.3 refines the initial (and rather abstract) goals **G1** and **G2** into concrete objectives (**O1** to **O4)** based on the analysis of the state of the art. Chapter 3 provides an overview of the contribution with respect to the concrete objectives **O1** to **O4**. Chapter 4 includes a commented collection of seven co-authored publications that form the core of the thesis. Chapter 5 presents the evaluation strategy followed, and Chapter 6 concludes the thesis and gives the author's subjective view of the promising research directions related to the area of model-driven development of siCPS with emphasis on dependability and self-adaptivity aspects.

Chapter 2

State of the Art

This chapter includes an overview of the state-of-the-art approaches that can be employed in the systematic analysis and design of siCPS.

Since siCPS is a relatively new class of systems with a number of distinct characteristics (as overviewed in Section 1.2), there is no approach that can be applied out of the box. At the same time, siCPS are inherently complex and dynamic systems exhibiting behavior that is difficult to predict and control (and sometimes emerges from the interaction between individuals – emergent behavior). In order to be able to design and develop applications for siCPS in a systematic way, we need to leverage ideas and approaches from existing methods that span different phases of the software development life cycle. In particular, a promising direction is to combine approaches that partially tackle the challenges of open-endedness, dependability and context-driven self-adaptivity and help reduce the development effort. Such approaches range from the domain of requirements modeling and analysis for safety-critical, open-ended and self-adaptive systems (Section 2.1.1), to multi-phase software engineering methodologies for agent-based systems (Section 2.2.1), to implementation abstractions for large, dynamic and distributed component-based systems (Section 2.2.2).

2.1 Requirements Modeling and Analysis for siCPS

The measure of success of a software system is the degree to which it satisfies the purpose for which it was intended. Software systems Requirements Engineering (RE) is the process of discovering that purpose, by identifying stakeholders and their needs, and documenting these in a form that is amenable to analysis, communication, and subsequent implementation [NE00]. It is within the scope of RE to identify, formulate, analyze, and agree on (i) what problem should be solved, (ii) why it needs to be solved, and (iii) who should be involved in the responsibility of solving that problem [VL09].

In spite of the differences in aim and supporting techniques, the activities composing the RE process are highly intertwined. They include the activities of domain analysis, requirements elicitation, negotiation and agreement, specification, specification analysis, documentation, and evolution [VL00, VL09]. Each activity has different challenges and supporting methods. For instance, in domain analysis, techniques from feature-oriented

software development [AK09] can be employed, such as the well-known Feature-Oriented Domain Analysis (FODA) method [KCH+90].

Throughout the RE processes, modeling is an important asset. The existing system/organization as well as the possible alternative configurations for the system-to-be are typically modeled. These models serve as a basic common interface to the various RE activities. They facilitate requirements elicitation by allowing looking at the domain systematically, and assist the communication between customers, analysts and developers. Formal models can be automatically checked for consistency, a process that can reveal conflicting and/or infeasible requirements. Models can provide a basis for requirements visualization, documentation, and evolution. Finally, following a Model-Driven Engineering (MDE) methodology, initial requirement models can be refined into architecture models and finally into code via subsequent model-to-model and model-to-text transformations.

Traditional requirement modeling techniques, such as structured analysis [RS77] and object-oriented analysis [AN05] have proven inadequate when dealing with more and more complex systems [MCY99, C09, Zav97]. When specifying such systems, it is crucial to view them as composites of the software-to-be together with its environment. Incorrect assumptions about the environment of a software system can lead to many errors in requirements specification [VLL04]. Non-functional requirements are also in general left outside of requirements specifications. Additionally, traditional modeling and analysis techniques do not consider alternative system configurations where more or less functionality is automated or different assignments of responsibility are modeled and compared. Goal-Oriented Requirements Engineering (GORE) [Kav02, MCY99, VL01] attempts to solve these problems by focusing on the "why" and "who", instead of the "what" and "how" questions of RE.

Although this thesis does not focus on the RE phase of siCPS development, nor on GORE itself, we overview next the prominent models and methods in GORE and identify key notions and ideas that can be employed in a holistic systematic design and development of applications for siCPS. The importance of these RE approaches lies in capturing the high-level behavior of a system and its stakeholders and providing this information as input to the subsequent design phases.

2.1.1 Goal-Oriented Approaches

The main abstraction in GORE is the *goal*. A goal can be defined as a prescriptive statement of intent about some system whose satisfaction in general requires the cooperation of some agents forming the system [VL04]. Goals are optative statements, as opposed to indicative ones (such as domain properties). Goals that have a clear-cut satisfaction condition are referred to as *hard-goals* or just goals; the opposite are usually referred to as *soft-goals*.

Another basic abstraction is the *agent*. Agents are active entities, i.e. entities with a choice of behavior. They may restrict their behavior to ensure the constraints that they are assigned. In GORE, agents are assigned responsibility for achieving goals.

There are a number of benefits associated with using goals for modeling and analysis in RE [VL01]:

- *Wider system engineering perspective.* Goals should hold in the system made of the software-to-be and its environment; domain properties and assumptions are explicitly captured during the requirements elaboration process, in addition to the usual software requirement specifications.
- *Rationale and traceability.* Goals allow for understanding requirements with respect to high-level concerns in the problem domain; they provide rationale for requirements that operationalize them. Goal refinement trees provide traceability links from high-level strategic objectives to low-level technical requirements.
- *Sufficient completeness criterion.* A requirements specification can be considered complete with respect to a set of goals if all the goals are proven satisfiable from the specification and the properties known about the domain.
- *Pertinence criterion*. A requirement is pertinent with respect to a set of goals if its specification is used in the proof of at least one goal.
- *Support for non-functional requirements.* The notion of soft-goals, which can be refined and analyzed, captures the rationale behind non-functional requirements and helps in their analysis.
- *Conflict management.* Conflicts among different stakeholder viewpoints and needs can be detected and managed early on, at the level of system goals.
- *Variability.* A single goal model can capture variability in the problem domain by alternative goal refinements and alternative assignment of responsibilities. Quantitative and/or qualitative analysis of these alternatives is possible.
- *Evolution.* Whereas traditional approaches "freeze" the solution decisions early on, goal-orientation separates stable (objectives) from volatile (requirements, tasks) information and thus provides the basis for system evolution.
- *Communication and documentation.* Goals provide the right level of abstraction to (i) communicate the requirements, and (ii) involve decision makers (e.g. customers, end-users) in the process of choosing between alternatives, proposing new alternatives, etc. Goal modeling provides also a natural mechanism for structuring complex requirements documents.

It is important to note that the goal-oriented requirements elaboration process ends where most traditional specification techniques (e.g. UML use cases [AN05]) would start [VLL04]. Thinking in terms of goals in the early phases of software engineering is a usual practice, as admitted by UML advocates [Fow03]; in GORE, this practice is just systematized.

2.1.1.1 KAOS

A prominent approach in GORE is Keep All Objects Satisfied (KAOS) [VLL04] (originally abbreviated from Knowledge Acquisition in AutOmated Specification [DVLF93]). KAOS was one of the first approaches, together with i* (Section 2.1.1.3), that advocated the use of goal as a main abstraction to drive requirements analysis. KAOS effectively combines goal-oriented principles with object-oriented and scenario-based ones into a comprehensive methodology for the elicitation, specification, and analysis (e.g. of completeness and pertinence) of requirements of virtually any software system.

The methodology includes the (i) goal elaboration, (ii) object modeling, (iii) agent modeling, and (iv) operationalization steps, optionally followed by the (v) conflict resolution and (vi) obstacle analysis steps. These steps are not performed in strict sequence; notably (i)-(iii) are typically intertwined. KAOS provides a rich meta-model supporting multiple views of the composite system (Figure 2), which comprises both the system-to-be and its environment.

A goal in KAOS is a declarative statement of intent to be achieved by the system under consideration [DVLF93, VL00]. The identification of relevant goals is the starting point of the methodology. Goals are identified by looking for intentional keywords in natural language statements used in requirements elicitation documents, and asking "why" and "how" questions about such statements. Goals in KAOS may refer to functional properties (services) or extra-functional properties (quality of services) and range from high-level to low-level concerns. Each goal belongs to one of satisfaction, consistency, safety, privacy, robustness, etc. categories [DVLF93]. Once the initial goals are identified, they are refined using AND- and OR-decompositions, forming goal decomposition graphs. A goal model is the collection of such goal decomposition graphs. (An example of a KAOS goal model is depicted in the paper included in Section 4.2.)

An agent in KAOS is an active component with certain capabilities that plays a specific role in goal satisfaction. An agent can be either a component of the system-to-be (e.g. software component) or its environment (e.g. hardware component, human user). KAOS agents refer to roles rather than individuals. Agent behaviors are modelled by interaction scenarios at the instance level and by parallel state machines at the class level [VL07]. The agent model defines the capabilities and interfaces of the various agents, and is devised in parallel with the goal model.

An object in KAOS is a passive entity that is being referenced by some goal in the goal model. An object is modeled as an entity, association, or event, depending on whether it is an autonomous, subordinate, or instantaneous object, respectively. The object model is graphically represented by a UML class diagram and is devised in an "on-demand" basis during goal elaboration, as new domain concepts need to be modeled so that they can be referenced by goals in the goal model.

An operation in KAOS is a specification of a service to be provided by the composite system. An operation is performed by an agent and is directly derived from goals [LVL02]. Operations are declared by their signatures (inputs and outputs) over objects, and have pre-, post-, and trigger conditions. KAOS distinguishes between *domain* and

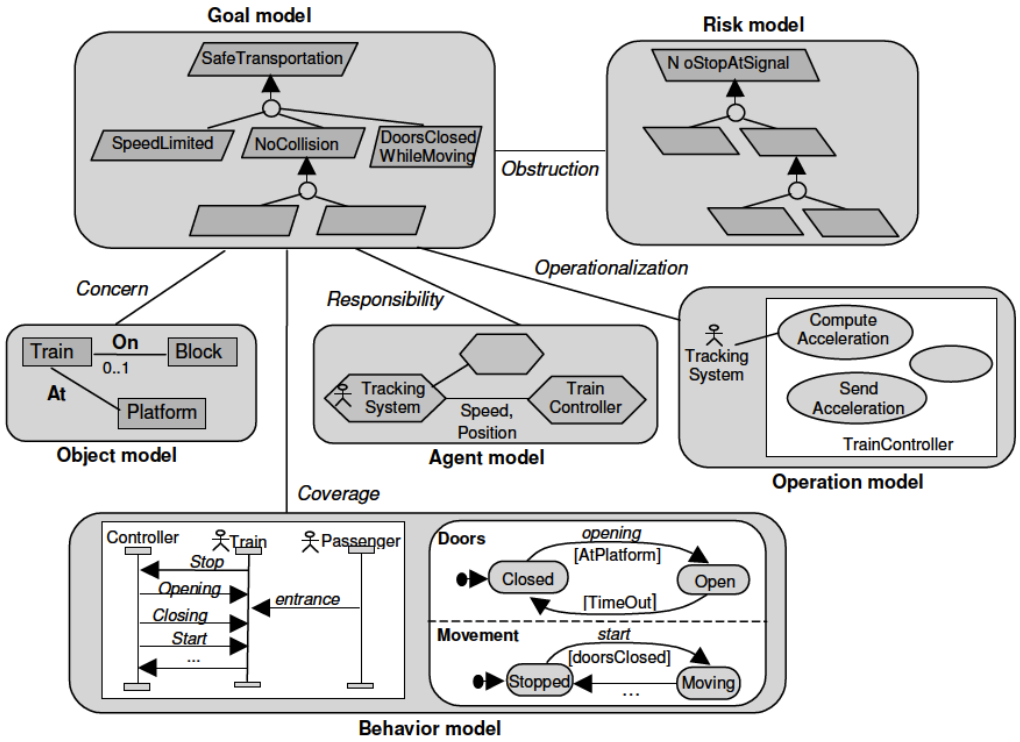


Figure 2. KAOS multi-view modeling (from [VL09]). Different models serve as a common interface to various RE activities.

desired pre-and post- conditions, the former capturing what the application of the operation means in the domain (indicative statements), the later capturing additional strengthening of the conditions to ensure that the corresponding goals are met (optative statements). An operation can be thought of as a use case in object oriented modeling.

An obstacle in KAOS is a concept that captures undesirable yet possible conditions [VLL00]. An obstacle obstructs a goal in the sense that, when an obstacle gets true the goal may not be achieved. Hazards and threats are obstacles obstructing safety and security goals, respectively [VL07]. Obstacle analysis is concerned with the resolution of obstacles via various techniques such as goal weakening, goal substitution, agent substitution, and obstacle mitigation ([VLL00] provides a detailed account on obstacle analysis techniques in KAOS). Obstacles are typically identified by negating a goal and refining the resulting obstacle via iterative AND- and OR-decompositions (as in goal elaboration), forming a fault tree. An obstacle model is a set of goal-anchored fault trees.

The aforementioned models are linked together in KAOS in the following way (Figure 2). First, iterative decomposition of goals is performed, during which elements of the object model are identified and modeled. Goals are associated with objects via "concerns" links. Goal decomposition ends once the leaf goals can be realizable by an agent assigned to it, i.e. the goal has to be expressed in terms of conditions monitorable and controllable by the agent. A leaf goal is a requirement or an assumption, depending on whether its satisfaction is assigned to an agent of the system-to-be or of the environment,

respectively. Leaf goals are associated with agents via "responsibility" links and with operations via "operationalization" links. Independently, scenario-based (typically represented by UML sequence diagrams) and state-based models are associated via "coverage" links with the goal model, while an obstacle model is associated with "obstruction" links with the goal model.

A distinct characteristic of KAOS is its support for formal specification and analysis. In fact, KAOS employs a "two-button approach": formal methods are used *when* and *where* needed. Specifically, goals and operations may optionally have a specification in real-time linear time logic (LTL) [Koy92], a first-order temporal logic. Although abstract goals near roots of trees cannot usually be directly formalized [DVLF93], lower-level goals and operations can be formalized in terms of behaviors that they generate or restrict. When goals and operations are formalized, the underlying models can be checked for completeness, minimality, and consistency, providing early validation of the requirements artifacts (particularly crucial in mission-critical systems [PMR+07]).

A semi-formal approach is also provided. For instance, goal definition patterns can be employed in KAOS for lightweight specification of goals at the modeling layer; they include patterns such as *achieve*, *cease*, *maintain*, and *avoid*, with the usual semantics in LTL [DVLF93]. In addition, KAOS provides a set of goal refinement and goal operationalization patterns [DVL96, LVL02]. A refinement pattern is a one-level AND-tree of abstract goal assertions such that the set of leaf assertions is a complete refinement of the root assertion. A catalogue of reusable patterns is intended to aid the analyst in the refinement process (e.g. by checking whether decompositions are complete refinements), while hiding the underlying mathematics. Generic patterns, proven correct once, can be reused via instantiation in a per-case basis. Similar to goal refinement and operationalization patterns, obstruction refinement patterns have been also proposed [VLL00].

2.1.1.2 NFR

Non-Functional Requirements (NFR) is a goal-oriented framework for justifying decisions during the software development process in terms of non-functional requirements (or qualities) [CNYM99, MCN92, MCY99]. NFR proposes a qualitative treatment of requirements (as opposed to a quantitative one) on the pragmatic basis that it is even harder to measure an *incomplete* system than a *complete* one in a quantitative way (and quantitatively measuring a complete system is already hard).

The main concept in NFR is that of a soft-goal. A soft-goal is said to be *satisficed* (an AI notion introduced in [Her96]), as opposed to *satisfied*, when it is met to an acceptable degree rather than absolutely [VL00]. Differently put, a soft-goal is satisficed when there is sufficient positive and little negative evidence for this claim, and unsatisficeable when there is sufficient negative and little positive evidence [MCY99]. Soft-goals model both generic non-functional requirements (e.g. accuracy, performance, maintainability) and application- and project-specific ones. They are categorized into NFR goals, satisficing goals, and arguments, depending on whether they belong to a requirements category (e.g. accuracy), a design decision category (e.g. validated by), or a formal/informal claim, respectively [MCN92].

Relations between soft-goals in NFR include decompositions and contributions. Top-level soft-goals are analyzed using AND- and OR-decomposition relations, forming soft-goal subtrees. Contribution relations associate soft-goals across subtrees; they can be positive or negative. This allows a designer to capture the fact that, e.g. the soft-goal of "Access other staff's files" is an offspring of the "flexibility" soft-goal, and contributes negatively to the "security" soft-goal. The outcome of the goal elaboration process is a (possibly cyclic) graph, amenable to qualitative analysis. The analysis is based on a labeling algorithm that assumes an ordering of labels and marks each node (soft-goal) as satisficed, denied, conflicting, or undetermined.

Analysis of non-functional requirements by means of soft-goals, as originally proposed in [MCN92], is complementary to analysis of functional requirements by means of (hard) goals (as, e.g. performed in KAOS). In this setting, contribution relations between goals and soft-goals are modeled and used in conflict analysis and prioritization among alternative requirements [MCY99].

2.1.1.3 i*

Contrary to KAOS that stands as a formal framework for requirements elicitation and analysis, the *i** modeling framework focuses on the early stages of RE that precede the specification activities [Yu95, Yu97]. *i** centers on the notion "distributed intentionality" (from which it also takes its name), i.e. on the intentions of the various agents of an information system. It combines knowledge representation techniques, agent-oriented principles, and goal-oriented principles into a modeling framework that can be used, apart from requirements engineering, in business process reengineering, organizational impact analysis, and software process modeling [Yu95].

The actor in *i** is a concept capturing system or environment agents (e.g. a software component, a human), roles, and positions. Agents are concrete actors with specific capabilities; roles are abstract actors embodying expectations and responsibilities; positions are sets of roles played by an actor [LYM03]. The concepts of goal, soft-goal, task, and resource are used in *i** in order to capture the intentions and processes at different levels of specificity. A goal is a condition or state of affairs in the world that an actor would like to achieve; a soft-goal is a goal whose satisfaction criteria are not clear-cut but subject to interpretation; a task is an activity carried out by an actor; a resource is an entity, physical or informational, provided by an actor. Tasks are more specific than (soft-) goals, as they specify a particular way – out of many possible – to satisfy the goal.

The framework consists of two models: the Strategic Dependency (SD) and the Strategic Rationale (SR) models, represented by SD and SR diagrams, respectively. (Examples of such diagrams are included in the paper of Section 4.2; an overview of the notation is available in [12]). In the SD model, the system actors and their dependencies are modeled in order to identify and explore the opportunities and vulnerabilities of each actor. The dependencies between actors in SD refer to goals that are expected to be satisfied (goal/soft-goal dependencies), tasks to be carried out (task dependencies), and resources to be furnished (resource dependencies). A goal dependency between actors *A* (depender) and *B* (dependee) models the fact that *A* relies on *B* for the satisfaction of the

associated goal, without however specifying *how*; this can be modeled by the task dependency.

In the SR model, the focus shifts from modeling the (external) dependencies between actors to modeling the rationale behind the (internal) processes within the boundaries of a single actor. A SR model captures the viewpoint of a single actor, and, as such, a separate SR model is "embedded" into each actor of the SD model (whose viewpoint is of importance). For each actor, the relevant goals, soft-goals, tasks, and resources are identified and associated with each other by task-decomposition, means-ends, and contribution relations. Task decomposition models the way that a task is AND-decomposed into an arbitrary number of tasks, goals, and resources. Means-ends models the way a goal is achieved, a task is performed or a resource is furnished (ends) via another task (means); it is typically used to model the variability in satisfying a goal. Contribution relations model the positive or negative contribution of goals, soft-goals, and tasks to the satisfaction of soft-goals; these relations are typically used for choosing the process configuration (prescribed by the SR model) that best meets the chosen non-functional requirements, in the spirit of the NFR framework (Section 2.1.1.2).

A distinct characteristic of *i** is that it helps reveal the organizational context of the system-to-be by focusing on the different and possibly conflicting intentions of the involved actors. Although the *i** meta-model is formally described in the Telos language [MBJK90], *i** does not focus on formal analysis. *i** models can still though be analyzed in terms of ability, workability, viability, and believability [Yu95].

Due to its generality and its rich set of concepts, *i** has been embedded in other frameworks. For example, it has been employed in a methodological framework for analyzing security and privacy requirements in order to supports dependency vulnerability, countermeasure, and access control analysis techniques [LYM03]. Most importantly, it has been employed for the early and late requirements phases of the Tropos methodology for agent-based systems, overviewed in Section 2.2.1.1.

2.1.1.4 FLAGS

Fuzzy Live Adaptive Goals for Self-adaptive systems (FLAGS) is an approach for specifying adaptation requirements, i.e. requirements that concern countermeasures to be taken when application requirements fail [BPS10]. FLAGS relies on the goal, operation, and object models of KAOS (Section 2.1.1.1) and extends them.

The main concept proposed by FLAGS is that of adaptation goal, a special type of goal that corresponds to a countermeasure to be applied when a "conventional" goal (e.g. à la KAOS) fails to meet its satisfaction criteria [BPS10]. To model such criteria, FLAGS introduces the notion of fuzzy goal, whose satisfaction, as opposed to a crisp goal, is the result of a membership function returning a value in the range [0,1] – the degree of satisfaction. Membership functions are elicited based on user preferences and typically have trapezoidal or triangular shapes. Fuzzy goals are not necessarily modeled as soft-goals, although the latter are strong candidates for fuzzification. For the formal specification of fuzzy goals, both a dedicated language (an extension of LTL with fuzzy operators) [BPS10, PS11], and the RELAX language have been employed [BP10a, BP11].

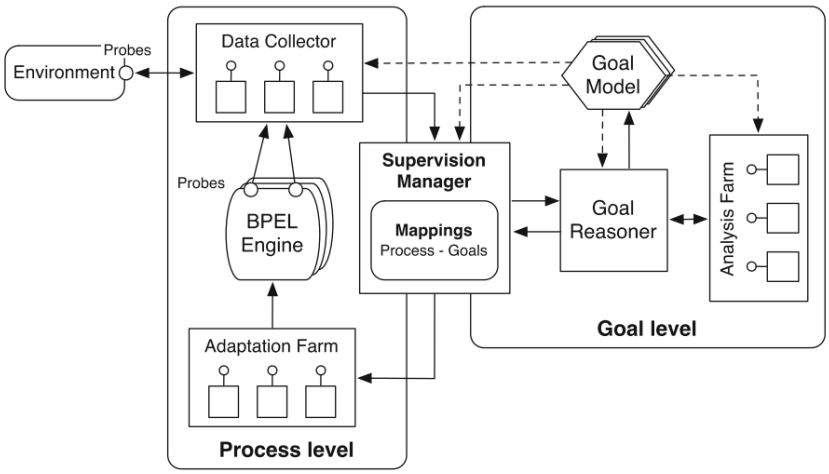


Figure 3. FLAGS runtime infrastructure (from [BP11]). Goals are kept as runtime entities; they trigger adaptations as countermeasures.

It is important to note that, contrary to KAOS and *i**, where variability in requirements is captured mainly by the use of alternative decompositions or means-ends relations, in FLAGS variability is captured via adaptation goals.

The FLAGS approach transcends the boundaries of requirements engineering by considering that goals (both conventional and adaptation ones) are kept as runtime entities that are continuously updated based on monitoring of the application and its environment (this is sometimes called "requirements reflection" [BWS+10, SBW+10]). According to the degree of satisfaction of goals at runtime, adaptation goals can be triggered, resulting into counteractions that include adding/removing goals, operations, or objects, modifying the membership function of goals, and adjusting the pre- and post-conditions of operations [BPS10].

A realization of the requirements reflection idea of FLAGS has been carried out in the domain of service compositions [BP10a, BP10b, BP11]. In the proposed runtime infrastructure (Figure 3), an existing service coordination engine, based on Business Process Execution Language (BPEL) [BBE+07], is instrumented to provide the necessary data for detecting events, updating the state of objects in the object model, and evaluating goal satisfaction in the "live" goal model. The satisfaction of goals is determined by pluggable analyzers of logic expressions (in LTL or their fuzzy counterparts) specified at the requirements phase. When needed, adaptation goals are triggered, which in turn trigger adjustments to the running BPEL processes. To achieve this, a mapping is kept between the runtime representation of requirement entities such as operations (linked to goals in the goal model), objects, and agents, and their counterparts in BPEL.

2.1.2 Lessons Learned

In this section, we attempt a critical examination of the advantages and limitations of the GORE approaches overviewed above in order to evaluate them in terms of their applicability in the analysis and design of siCPS.

A first observation is that the goal abstraction has proven particularly useful in GORE; goals have been used to model functional requirements (KAOS), non-functional ones (NFR), and both functional and non-functional requirements in the same model (i*). Goal-orientation points to the right direction of thinking about the reasons behind the requirements elaboration and documenting them during the elaboration process. Relying on goals and other GORE abstractions (e.g. actors' intentions, dependencies, plans) during the subsequent architecture design phase, where design decisions need to be captured together with their rationale, seems promising. Such an approach has been attempted in Tropos, an agent-oriented methodology overviewed in Section 2.2.1.1.

Reflecting on the goal classification schemes used in GORE, and in particular KAOS that distinguishes between achieve/cease and maintain/avoid goals, the *maintain/avoid* goals seem to be more pertinent to the domain of siCPS. The high degree of dynamicity and operational uncertainty in siCPS thwarts the application of multi-step end-to-end protocols that typically underlie the achieve/cease goals. On the contrary, the notion of "striving to achieve", manifested also in novel RE methods for dynamic self-adaptive systems, such as SOTA [ABZ12], fits better the domain of siCPS, as computation is typically targeted at maintaining a form of *operational normalcy*, i.e. the property of being within certain limits that define the bound of normal operation. The notion of operational normalcy roughly corresponds to the maintain/avoid goals in GORE.

Another GORE notion pertinent to the analysis and design of siCPS is that of fuzzification of goal satisfaction criteria, as proposed in FLAGS and in other approaches (e.g. RELAX [WSB+10]). Being able to model and reason about properties in a fuzzy way provides a powerful mechanism to deal with uncertainty. Uncertainty generally manifests itself in various SE phases (during requirements elicitation, architecture design, deployment, and others), takes various forms (from imprecise measurements to probabilistic execution), and stems from various reasons (from vagueness in requirements coming from stakeholders to unpredictable situations in the environment). In siCPS, there is a particular need to address uncertainty stemming from the unpredictable and ever-changing environment (sometimes called *external* uncertainty [EKM11]). Employing fuzzy models is a way to embed external uncertainty in system specification, which is a prerequisite for dealing with it via other means in the next phases (e.g. via switching to fail-safe modes at runtime in case high external uncertainty is present).

Checking the validity of requirements is also another recurrent topic in GORE, which is also relevant to the analysis and design of siCPS. As overviewed before, requirements checking can be done both at design time and at runtime, with complementary methods. While at design time the focus is on checking the completeness and pertinence of the whole set of requirements to high-level goals (as done e.g. in KAOS), at

Table 1. Main outcomes of overviewed GORE approaches.

GORE approach	Main outcome
KAOS	Software Requirements Specification (SRS) document [16]
NFR	Goal graph that justifies design decisions in the development process according to the degree to which non-functional requirements are satisfied
i*	A set of goal graphs representing the strategic dependencies and the rationale of actors in the system
FLAGS	KAOS-like goal and object graphs (where goals are formalized in fuzzy logic), augmented by a set of adaptation goal graphs

runtime the focus is on checking whether individual requirements are satisfied – and to which degree. In particular, the idea of considering requirements as runtime entities, as proposed in FLAGS and other works advocating requirements reflection (e.g. [BWS+10, SBW+10, SLRM13]), is particularly suitable for requirements checking in siCPS, as any form of design-time checking typically relies on strong assumptions about the environment which are not plausible in most siCPS.

A final important remark is that the focus of KAOS, i*, NFR, and FLAGS, but also of other RE approaches such as ALBERT [DBP94], GBRAM [Ant96], GRL [AGH+10], RELAX [WSB+10], is at the requirements analysis and not at the architecture design phase. As summarized in Table 1, the main outcomes of these approaches are a set of requirement specifications in both graphical and textual form, amenable to different forms of analysis ranging from informal stakeholder inspection to formal verification via model checking. While focusing on requirements analysis is not a limitation per se, these approaches do not provide a systematic way to translate the requirements artifacts to architecture and implementation constructs, in order to bridge the well-known gap between requirements and architecture [AGH+11, BKW03, KC01]. In this direction, notable attempts have been made in the context of KAOS (e.g. in [VL03]) and i* (e.g. the Tropos methodology, overviewed in Section 2.2.1.1). FLAGS also proposes a manual mapping of goal models to functional models represented by BPEL processes [BP10b].

As the essence of an effective architecture design is precisely this mapping between requirements and architecture, we focus next on promising methodologies that try to systematize this in the domain of dynamic distributed systems akin to siCPS.

2.2 Software Development Methodologies and Implementation Abstractions for siCPS

A software development methodology is a process that guides software development through different phases, which prescribe different activities. Requirements engineering, and in particular GORE (Section 2.1.1), is typically one of the first phases, followed

by architecture design, implementation, testing, deployment, and maintenance. A distinction is often made between methodological models that focus on the development life cycle (e.g. spiral [Boe00], Unified Process [AN05], agile [SW07], MDE [Sch06]), which are considered as broad classes of methodologies, and specific instances of development processes (e.g. Scrum [Coh09], Extreme Programming [Bec99], DARTS [Gom93]) that typically belong to one or more of the broad classes.

Using the same set of concepts across phases is generally considered an asset, as it allows for easier mapping of artifacts (e.g. requirements specifications, architecture diagrams, code) from one phase to the next. Hence, each specific process (refinement of a generic methodology) is typically characterized by the paradigms used for modeling and implementation. For example, although UML [AN05, Fow03] is considered general-purpose, it primarily follows the object-oriented modeling paradigm; this, in turn, characterizes also the development processes that employ UML as "object-oriented".

Since siCPS are complex and dynamic systems, we are interested in paradigms, implementation abstractions, and corresponding methodologies that can deal with the complexity of building and maintaining large systems, while accounting for dynamic and unpredictable environments. Promising paradigms are the agent-oriented and the component-based modeling and development paradigms, overviewed in Sections 2.2.1 and 2.2.2, respectively.

2.2.1 Agent-Oriented Software Development

In an effort to deal with the increasing complexity of software systems that have to operate in dynamic, unpredictable, and heterogeneous environments, Agent-Oriented Software Development (AOSD) has been put forward [SS14a, WC01]. AOSD views systems as organizations of self-governed entities, called *multi-agent systems* [SLB08], with independent life cycles and loci of control; this viewpoint is a perfect fit when modeling complex distributed and decentralized systems [Jen00].

The main abstraction in AOSD is a software *agent*. An agent is an encapsulated software system that is situated in an environment and is capable of autonomous action in order to meet its design objectives [Woo97]. The properties commonly attributed to an agent is that of autonomy, reactivity, proactiveness, and social ability [Woo97]. The conceptual autonomy is a main difference between agents and objects; they both encapsulate state and behavior, but agents have additional control over behavior activation and action choice (they are *active* entities in contrast to *passive* objects). From a different viewpoint, agents provide a higher level of abstraction independent of the implementation technology, be it objects or components (discussed in Section 2.2.2) [Jen00].

A prominent reference model for AOSD is Belief-Desire-Intention (BDI) [Bra99]. Agents in systems architected according to the BDI model (so-called *BDI agents*) possess goals (desires), plans (intentions) and knowledge about the world (beliefs) [RG95]. Goals are the objectives a BDI agent pursues, plans are the means of achieving certain future states, and beliefs (or belief base) is the knowledge a BDI agent possesses about the environment, including the knowledge about other agents. BDI agents follow a reasoning

cycle where they continuously (i) monitor the environment, (ii) decide on which goals to achieve, and (iii) activate the plans that are most likely to lead to achievement of their goals. This reasoning cycle complies with the Monitor-Analyze-Plan-Execute (MAPE-K) reference model for autonomic computing [IBM06, KC03]. There exist various implementations of the conceptual BDI architecture model, based on pure Java (e.g. Jadex [PBL05][18], JACK [Win05][17]), combination of Java and Prolog (e.g. 3APL [HBHM99]), and first order logic (e.g. AgentSpeak [Rao96], and its Java interpreter, Jason [BH06]).

The new abstractions introduced by the agent-oriented modeling paradigm call for tailored analysis and design methods that perceive the system as a collection of intentional actors with social abilities. Several AOSD methodologies have therefore emerged to bridge this gap: Tropos [BPG+04], Gaia [WJK00], MaSE [DWS01], O-MaSE [DeL14], ADELFE [BGPP03], Prometheus [PW03], IGNENIAS [GS14], etc. Such methodologies provide guidelines and methods that support AOSD from requirements engineering, where goal orientation (Section 2.1.1) is typically favored, to architecture design, down to detailed design and implementation (typically in a BDI platform) [DW04].

A comprehensive review of the state of the art in AOSD methodologies is available in [SS14b]. In the rest of the section, we delve into the specifics of a representative AOSD methodology.

2.2.1.1 Tropos

Tropos is a well-known holistic approach towards the development of information systems perceived as agent-based software systems [BPG+04, CKM01, CKM02, GKMP04]. Tropos combines the agent-oriented modeling paradigm with goal-orientation in requirements into a comprehensive methodology that spans all of the main software development phases, from requirements elicitation and analysis to implementation. It provides guidelines on how to align these phases using concepts from early requirements engineering such as actors, goals, and dependencies [CKM02]. At the same time, its mentalistic notions are founded on BDI agent architectures [Bra99, RG95].

Tropos methodology spans five phases: early requirements analysis, late requirements analysis, architectural design, detailed design, and implementation.

In the early requirements analysis phase, the organizational setting of the system-to-be is studied and modeled, focusing on the stakeholders and their intentions. For this phase, Tropos adopts and *i** modeling framework (Section 2.1.1.3) and suggests building a strategic dependency (SD) and possibly several strategic rationale (SR) models. As described in Section 2.1.1.3, a SD model captures the social actors (which represent the stakeholders of the system-to-be) and their interdependencies in terms of goals/soft-goals to be achieved, tasks to be completed, and resources to be furnished. A SR model, on the other hand, captures the goals, soft-goals, resources, and tasks of a *single* actor, together with their means-ends and AND/OR decomposition relations and soft-goal contributions. It also captures how dependencies that enter or exit the actor boundaries are contributing in fulfilling this actor's goals. Extending *i**, Tropos provides a formal language for specifying SD and SR models (also called *actor models* and *goal models* respectively in the context of Tropos [BPG+04]) termed Formal Tropos [FLM+04, FPMT01].

Formal Tropos allows the analyst to specify constraints on the models' elements in LTL that allow for consistency checking; in many ways it is close to the formal language of KAOS (Section 2.1.1.1).

In the late requirements analysis phase, the goals and soft-goals are operationalized resulting into a complete specification of the system-to-be together with its organizational setting. This phase takes as input the SD and SR models of the previous phase and extends them by adding an actor (potentially decomposed into sub-actors) representing the system-to-be, and modeling its dependencies to/from other actors. The revised SD and SR models should also contain the final specification of non-functional requirements (NFRs), after negotiation with stakeholders, by means of positive or negative contributions to soft-goals. The soft-goal analysis follows the NFR framework (Section 2.1.1.2)

In the architecture design phase, the overall system structure is captured in terms of subsystems interconnected with data and control flows. As in the previous phases, subsystems are captured as actors in SD and SR diagrams. This phase can be broken down into the following steps.

First, one or more organizational architectural styles are selected and instantiated in the design of the system-to-be. These include the styles of joint venture, flat structure, structure-in-5, pyramid, cooperation, and others [KCM01, KGM01, KGM06]. Unlike classic architecture styles (e.g. pipes and filters, layered systems, etc. [GS93]), they are inspired from research in organization management and specifically focus on the domain of cooperative, distributed and dynamic applications, such as multi-agent systems. Each style supports some and hinders other NFRs, such as security, adaptability, modularity, etc. The style that best supports the NFRs that resulted from the soft-goal analysis in the previous phase is chosen. For example, If "adaptability" and "integrity" were identified as important NFRs for the system-to-be, then an architectural style that supports them can be chosen, e.g. the joint venture [CKM02] in this case. The instantiation of the chosen style can result into re-arrangement of the actors in the SD model, introduction of new actors, removal of old ones, etc. Note that all these changes are done at a macro level, i.e. they aim to satisfy the overall goals of the system-to-be, not of a particular goal of a specific actor.

Second, for each actor, the capabilities needed to fulfill its goals and tasks are identified by analyzing the in-going and out-going dependencies in the SD model, similar to identifying API for classes by analyzing the sequence diagrams in object-oriented design. Then, each actor is mapped to one or more agents. This manual task requires a lot of expertise. To aid the designer, Tropos provides a set of social patterns that, unlike organizational styles, focus on the social structure necessary to achieve *one* particular goal [FGKM01, KGM01, TMA03]. These patterns are recurrent in multi-agent and cooperative systems; they include patterns such as broker, embassy, mediator, and others. Each pattern defines a set of capabilities associated with the agents involved in the pattern. The application of a pattern can result in introducing new agents (e.g. an intermediary agent in the mediator pattern) and, in general, guides the mapping of actors to agents in a systematic way, much in the same way as popular object-oriented design patterns systematize best practices in object-oriented systems [GHJV94].

In the end of the architecture design phase, revised SD and SR models are created, taking into account changes from instantiating the chosen organizational styles and social patterns. Also, a mapping between actors and concrete agents with specific capabilities is created.

The detailed design phase that follows is concerned with the specification of the agents' micro level, i.e. the agents' behavior and communication. This step is guided by the underlying implementation abstractions and corresponding platforms. Since Tropos is primarily intended as a methodology for building agent-based systems, a subset of the Agent UML [BMO01, OPB00] is typically adopted. Specifically, AUML class diagrams are used to capture the structure of actors in the traditional object-oriented way, where each actor is mapped to a class with attribute and method compartments. Additionally, AUML sequence diagrams are used to capture the interaction protocol between specific actors. Capability and plan diagrams (graphs that correspond to directly executable prescriptions of how an agent should behave to achieve a goal or respond to an event [KG97]) are used to specify the internal processing needed to carry out a specific task [CKM01, CKM02]. Note that a plan, in this context, is a refinement of a conceptual task included in the SR models. Classical UML activity diagrams have also been proposed within Tropos for capability and plan modeling [BPG+04].

Finally, in the implementation phase, the detailed design models are used to generate executable code in the target BDI platform. Both JACK Intelligent Agents [Win05][17], and Jadex [PBL05][18] have been used as target platforms. The agents specified in the previous phases and their capabilities, including plans, beliefs, and events to be handled, are mapped to JACK constructs [BPG+04] and Jadex constructs [PPS+07] in a rather straightforward way.

Tropos has been extended in many ways and the research around it followed different paths [31]. Next, we describe an extension that we deem important for the context of this thesis, as it focuses on context-aware self-adaptation in agent-based systems.

2.2.1.2 An Extension of Tropos for Adaptive Systems

A pragmatic approach towards extending Tropos and applying the extension in the tool-supported development of self-adaptive agent-based systems is *Tropos for self-adaptive Systems* (Tropos4AS) [Mor11, MP08, MPP08a, MPP08b]. Tropos4AS introduces a number of extensions to the modeling layer of Tropos (and a clear mapping to concepts already featured in BDI languages such as Jadex) in order to allow designers model self-adaptive behavior by capturing failure conditions and corresponding countermeasures at the level of goals and tasks [MP08]. The extensions include:

- Specification of goal types according to their satisfaction criteria as *achieve goals* (to be satisfied only once in the lifetime of the agent), *maintain goals* (to be satisfied continuously), and *perform goals* (without concrete satisfaction criteria) – similar to goal patterns in KAOS;
- A goal prioritization mechanism based on inhibition relations between goals;

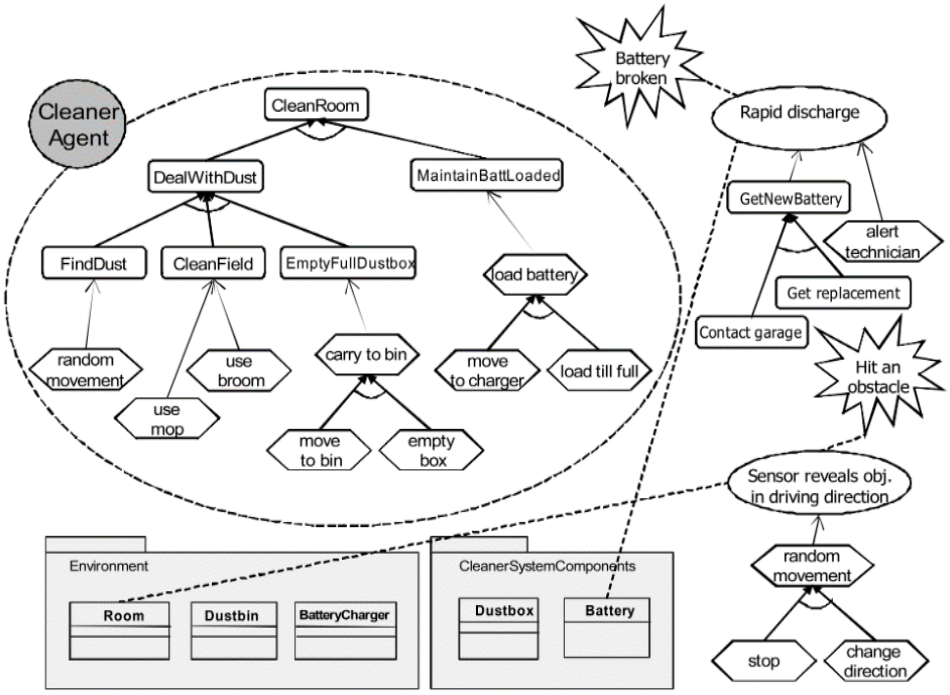


Figure 4. A Tropos4AS model (from [MP08]). Rounded rectangles denote goals, hexagons tasks, jagged circles symptoms, and ellipses errors.

- Modeling of the non-intentional entities of the environment (environment entities) via standard UML class diagrams – similar to object modeling in KAOS;
- Specification of *conditions* for goal creation and satisfaction that associate goal model entities with environment entities;
- Modeling of *errors* (or *failures*) and corresponding *symptoms* (attached to environment entities) that represent failure conditions;
- Modeling of recovery activities (attached to symptoms) by means of new goal models complementary to the main goal model of each agent.

Figure 4 depicts an example of a Tropos4AS model. A mapping is provided between the requirements-level concepts captured at the goal model and the implementation-level concepts available in Jadex. In summary, actors are mapped one-to-one to Jadex agents, actors' main goals and goals that correspond to recovery activities to Jadex goals, tasks to agent's plans, and resources and environment objects to facts in the agents' belief bases. The different goal types (maintain, achieve, and perform) and the inhibition relations between goals are mapped to corresponding concepts already available in Jadex. Also, means-ends relations are mapped one-to-one to the Jadex plan triggering mechanism (in the sense that activated goals are triggering events for plans). AND, resp. OR, decompositions are mapped to sub-goal dispatch algorithms that activate and monitor the execution of all, resp. one, of the subgoals of an activated parent goal, as specified in

the non-leaf goal semantics [MPP09]. Finally, dependency relations are mapped to plans on both the depender and the dependee agent that realize a FIPA-standard communication protocol [ON98] and rely on Jadex agent messaging functionalities. In order to allow for advanced decision-making at runtime, the goal model is also stored in the agents' belief bases. The above mapping is supported by a dedicated tool that provides code generation to XML-formatted Agent Definition File (ADF) and Java [MPP08a].

Self-adaptation in Tropos4AS is pursued both by choosing among alternatives in OR-decompositions and between different means-ends relations attached to the same goal, and by activating goals that correspond to recovery activities. Once a goal is activated, the "best" plan associated with its satisfaction is triggered, which leads to a series of actions performed by the agent. Changes in the agent's belief base may lead to the activation of new goals, which trigger new plans, and so on.

2.2.2 Component-Based Software Development

Managing the complexity of building and maintaining large software systems is a challenge. Component-Based Software Development (CBSD) is a widely adopted approach to tackle this challenge primarily by relying on separation of concerns, decomposition, and systematic reuse of existing artifacts [CL02]. CBSD adopts an architecture view over software systems and provides abstractions for software composition, interoperability, and reuse.

The main abstraction in CBSD is a software *component*. A component is a software unit with contractually specified interfaces that are either required or provided; it encapsulates and generalizes the popular object abstraction found in object-oriented languages [CL02, Szy02]. A component can be developed and deployed independently and is subject to composition by third parties [HC01].

Components in CBSD adhere to a *component model*, which provides a set of rules (or standards) for component behavior, composition and interaction [CSVC11, LW07]. A component model, together with a corresponding *runtime platform* (sometimes referred to as *execution environment*) and related tools supporting design, development, and deployment, are often referred to as *component system* [CL02, Mal12].

Component systems – both industrial and academic, general-purpose and specialized – have been proposed and used in enterprise applications (e.g. CCM [24][WSO01], EJB [25], Sofa 2 [BHP06], Fractal [BCL+06]), configuration platforms (e.g. Spring [30], OSGi [27][HPMS11], Google Guice [11]), and embedded and real-time systems (e.g. MyCCM-HI [BHP09], Koala [VOVDLKM00], Sofa HI [PWT+08], ProCom [SVB+08], RTSJ component model [PMS08]), among other domains. A comprehensive overview and classification of existing component systems is provided in [CSVC11].

Table 2. A sample of EU projects targeting architecture modeling of CPS.

Project name	Project description
Smart Objects For Intelligent Applications	SOFIA is a glue platform for sharing interoperable information in smart environment applications. The platform can be discovered and accessed as a service (e.g. as NoTA, Web, or OSGi Service). Context-awareness, access control and security may be added in this way. The platform is agnostic with respect to ontology, programming language, service and communication levels, as well as hosting device/system.
ARROWHEAD	ARROWHEAD aims at addressing technical and applicative challenges associated with cooperative automation by providing a service-oriented framework for integration and cooperation between legacy systems acting as service providers and consumers.
Adaptive Cooperative Control in Urban (sub)Systems	ACCUS aims at the exploitation of tools that are available for distributed control systems, e.g. for event-based synchronization. Urban systems are considered to be a CPS which is controlled as a networked multi-agent system. The project addresses structural runtime adaptivity for CPS, and proposes a design methodology for emerging system-level behaviours.
DEsign, Monitoring and Operation of Adaptive Networked Embedded Systems	DEMANES aims to provide component-based methods and tools for development of runtime adaptive systems, making them capable of reacting to changes in themselves, in their environment (battery state, availability and throughput of the network connection, availability of external services, etc.) and in user needs (requirements).
Autonomic Service Component ENSembles	ASCENS focuses on analysis, design and development of systems of ensembles, i.e. large complex systems of collaborating entities that form dynamic groups. It features three case studies from the domains of electric vehicle mobility, ad-hoc cloud deployment, and smart robotics.

Attempts for modeling CPS in a component-based fashion in different application areas have been pursued at a European level within Artemis/ECSEL and FP7/H2020 research frameworks (Table 2 provides a summary of related projects). Indicative of the importance of the field, CPS has been identified as one of the "essential capabilities" in the ECSEL roadmap [9].

In the rest of the section we overview three component systems that specifically target large, dynamic, distributed, and software-intensive systems akin to siCPS. Compared to approaches found in different EU research projects (Table 2), which mostly focus on deployment by providing component- or service-based runtime platforms, the component systems reviewed next stand as representatives of a more systematic CBSD approach that relies on well-defined component models and tooling support for modeling, development, and deployment.

2.2.2.1 Kevoree

Kevoree [21] is a component system that relies on the paradigm of models at runtime (models@run.time). This paradigm proposes to use a self-representation of a system that

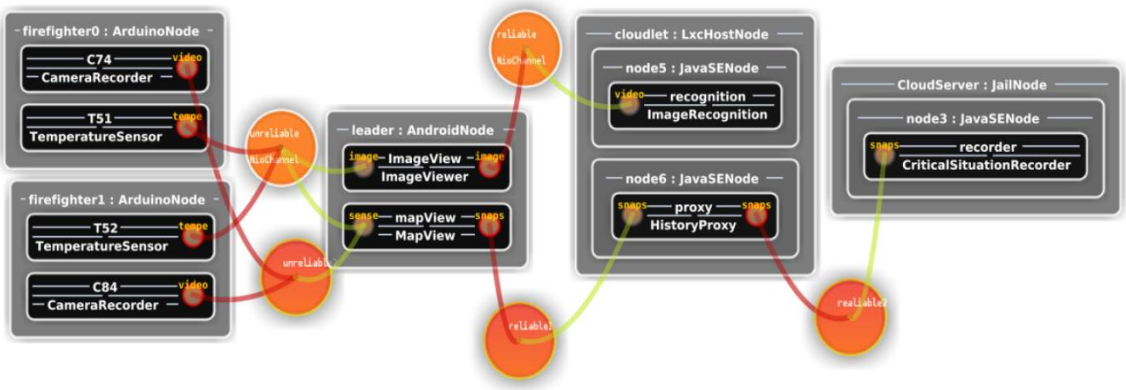


Figure 5. Example of an architecture created with the Kevoree graphical editor (from [AAGGH+14])

emphasizes the structure, behavior, or goals of the system from a problem space perspective in order to provide a reflection layer that is uncoupled from, yet causally connected to, the running system [BBF09, BFCA14]. Models@run.time is envisioned to endow systems with dynamic state and behavior monitoring, semantic integration, automatic generation of artifacts at runtime, and design evolution capabilities, among others [BBF09]. In the case of Kevoree, the employed self-representation emphasizes the architectural aspects, i.e. the architecture model is considered a runtime entity that is casually connected to the running system [MBJ+09].

The component model of Kevoree includes the concepts of component, channel, node, and group. A Kevoree component is a traditional CBSD component, i.e. a unit of development and execution with contractually specified interfaces. In Kevoree, interfaces are specified as input and output ports. A Kevoree channel is a traditional CBSD connector, i.e. a unit that realizes the communication between components according to some coordination style (e.g. synchronous, asynchronous, sequential, etc.). A Kevoree node represents a unit of deployment and stands as a container for components. Finally, a group is special concept used in Kevoree to refer to set of nodes that share the same (architecture) model, and the way this model is synchronized across the nodes. The idea is that, once one node of the group applies a reconfiguration strategy (defined in a dedicated language – KevScript [FBP+12]) and changes its architecture (reflection) model, the updated model is propagated and synchronized across all the nodes in the same group. With its concepts, especially the node and group ones, Kevoree provides proper support for distributed models@run.time [FBP+12].

Apart from a component model, Kevoree provides several execution environments (including JVM, Android, and Arduino) and mappings to several mainstream implementation languages, in particular Java, C++, and Javascript [21]. It also provides tools supporting design, development, and deployment. For instance, a graphical editor can be used for specifying the initial architecture (components and channels connecting them) and deployment (assignment of components to nodes and of nodes to groups) – an example is depicted in Figure 5.

2.2.2.2 Helena

Helena is a component-based approach for developing large distributed dynamic systems comprised of groups of entities that collaborate to achieve some common tasks [HK14, KH14, KMH14]. The Helena approach adopts the core ideas of a component model (e.g. separation of application logic into components and communication logic into connectors) and combines them with role-based modeling concepts [HK14]. Helena was conceived and developed in the frame of ASCENS FP7 project [1].

The main abstractions of Helena are components, roles, role connectors, and ensembles [HK14]. A Helena component is specified by its name, attributes, and supported operations. The operations are distinguished into incoming/outgoing (corresponding to required/provided interfaces) and internal. The outgoing and incoming operations are mapped to sending and receiving of messages. Although they may have operations, components in role-based approaches (including Helena) are usually passive entities (data containers), and serve as execution platforms for roles. A Helena role provides context-specific data and behavior; it is specified via its name, attributes, operations (outgoing, incoming, internal), and set of nodes that can play the role, i.e. exhibit the behavior of the role. Finally, a role connector captures the communication between roles in much the same way as a classic CBSD connector captures the communication between CBSD components [IST11]. A role connector specifies which role is the source of the communication, which is the target, and what message should be communicated (captured as an output operation of the source that matches an input operation of the target).

Having the concepts of components, roles, and role connectors in place, Helena proposes to models ensembles, i.e. goal-oriented groups of components. From the structural perspective, an ensemble is modeled as a set of roles, each with a multiplicity, and a set of role connectors. From the behavioral perspective, it is modeled in terms of role behaviors. Each role behavior is a labeled transition system that specifies the sequences of operations that can be executed on the role so that the role contributes the required responsibilities to the ensemble.

The whole approach is backed up by a Java runtime platform for execution of ensembles called jHelena [KH14]. Tool support for specification of ensemble-based architectures in a domain-specific language and subsequent code generation in Java (to be ran in jHelena) is also provided [KCH14].

2.2.2.3 DEECo

Dependable Emergent Ensembles of Components (DEECo) is a component system that has been proposed for the development of highly dynamic and distributed cyber-physical systems [BGAA14, KBPK12, Kez14, BGH+13]. DEECo combines ideas and concepts from CBSD (e.g. rigorous component model), agent-oriented computing (e.g. entities with autonomy), and real-time and control systems (e.g. periodic computation) into a comprehensive synergy. Like Helena, DEECo has been conceived and developed in the frame of ASCENS FP7 project [1].

The main abstractions in the DEECo component model are the component and the ensemble [BGH+13]. A DEECo component is an independent unit of development and

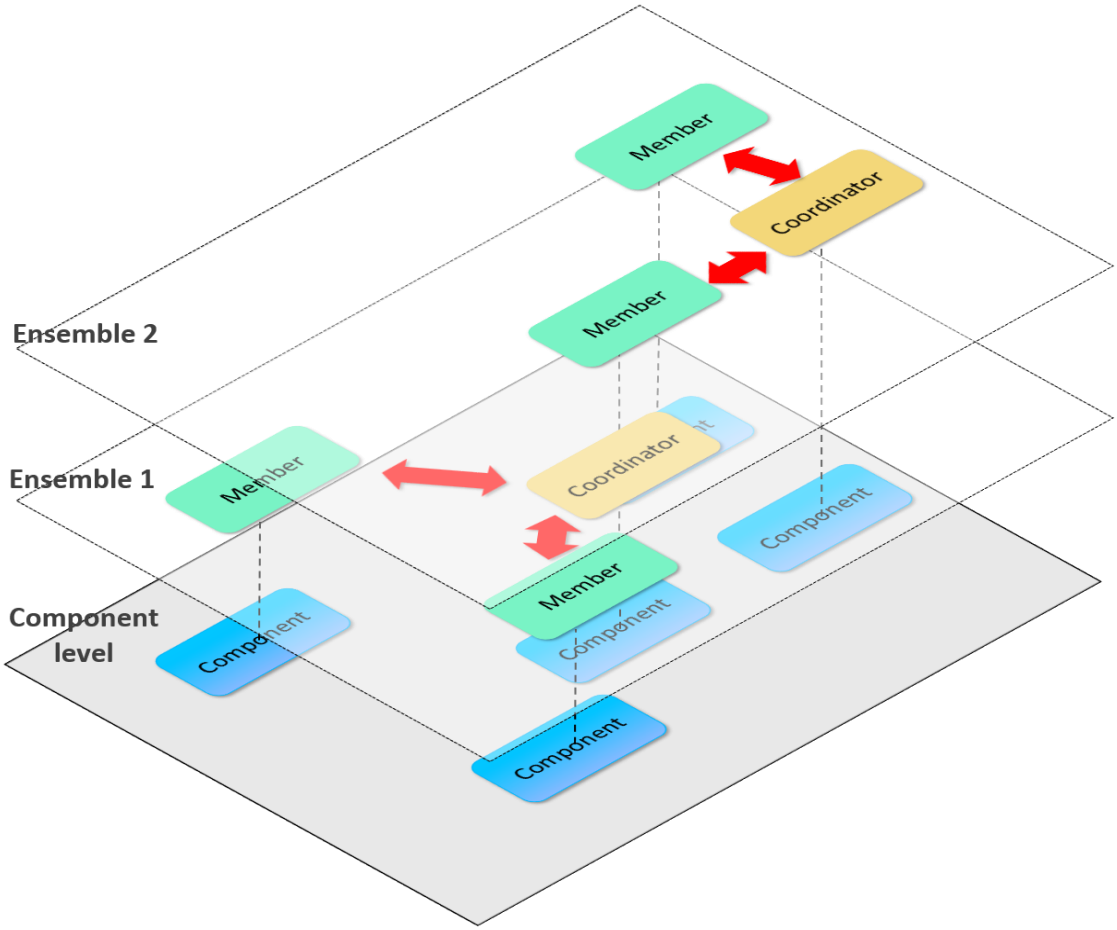


Figure 6. DEECo components can belong to multiple ensembles at the same time.

deployment that consists of state, modeled as knowledge, and functionality, modeled as processes. Knowledge is a hierarchical (tree-like) data structure mapping identifiers to (potentially structured) values. Values are either statically typed data or functions. DEECo employs statically typed data and pure functions (without side effects) as first-class entities. Each of them being essentially a thread, processes operate upon the knowledge of the component. A process employs a function from the knowledge of the component to perform its task. As functions are assumed to have no side effects, a process defines a mapping of the knowledge to the actual parameters of the employed function (input knowledge), as well as a mapping of the return value back to the knowledge (output knowledge). A process can be either event-or time-triggered (periodic). An event in this context is a change either in the valuation of input knowledge or in the satisfaction of a condition on component's knowledge. Contrary to conventional CBSD components, DEECo components are *autonomous* in the sense that they do not have required and provided interfaces, and do not bind with each other to form composites. Instead, each component provides part of its knowledge to the outer world (i.e. the other components); this knowledge then serves as communication medium.

A DEECo ensemble is a group of components that cooperates to achieve a common goal. It is dynamically established and disbanded at runtime depending on the state of the environment and the state of the components. A component can simultaneously belong to many ensembles, modeling the case when a component pursues more than one goals at the same time (Figure 6). Within an ensemble, components communicate indirectly via exchanging knowledge. An ensemble is specified by a *membership* condition and a *knowledge exchange* function. Membership specifies which components to involve in the ensemble, while knowledge exchange specifies which knowledge should be exchanged between these components. Instead of directly referencing components in an ensemble specification, a role matching mechanism is used: the ensemble roles of *coordinator* and *member* are specified; membership and knowledge exchange are then specified as a condition over the knowledge of coordinator and members, and assignment from the knowledge of members to the knowledge of coordinator (or vice versa), respectively. Component roles, i.e. sets of component knowledge, are also specified for each component and then dynamically matched to ensemble roles for each instantiated ensemble. Similar to a DEECo process, an ensemble is event- or time-triggered. When triggered, the ensemble first evaluates the membership condition and then executes the knowledge exchange for each member-coordinator pair.

Apart from the component model, DEECo comes with a runtime framework and corresponding mapping in C++ [2] and Java [19]. While the former targets actual deployment on embedded devices, the latter is primarily intended for quick prototyping and experimentation via simulations [KGB+15]. DEECo runtime frameworks provide the necessary utilities for (i) developing and deploying components and ensembles, (ii) scheduling of component processes and ensemble evaluations, (iii) managing the knowledge of each component (including performing knowledge exchange when needed). The Java implementation, in addition, provides support for rather detailed simulations comprising network-accurate communication and agent-based environment representation [KGB+15].

2.2.3 Lessons Learned

In this section, we reflect on the advantages and limitations of agent- and component-based modeling paradigms, provide a critical evaluation of the overviewed approaches, and justify our decision to adopt the abstractions proposed in DEECo as our implementation substratum for siCPS.

The agent-based modeling paradigm provides an effective set of concepts to model complex interactions between a large number of software entities in a flexible and intuitive way. The behavior of each agent can be programmed independently, relying on the agent's perception of the world and with compliance to the agent's goals. Agents' interactions are also conveniently mapped to massaging between agents supported by underlying service-based platforms that provide the utilities (e.g. service registry) for dynamic discovery and addressing. This adds to the conceptual autonomy of each agent and provides clear separation of concerns in the modeling and implementation phases.

However, we notice two drawbacks in the application of the agent-based paradigm in the design and development of siCPS. First, the interactions among the individual agents in a multi-agent system give rise to joint behavior that cannot be predicted a priori, let alone modeled and controlled (this is sometimes called *emergent behavior* [WHS06, Wol07]). This effect is not necessarily negative in case of large-scale environment simulations (e.g. mobility of vehicles and humans in a city), where agent-based approaches help in the identification of emerging trends and interactions. However, when developing large-scale siCPS, the focus is on eliminating – or at least bounding – the emergent behavior in order to provide a predictable and thus dependable system.

The second limitation concerns the implementation of agent-based platforms. As they are realized as service-oriented architectures (SOA) [PH07, WH11], they inherit both the advantages of SOA, namely third-party ownership and deployment, and dynamic architecture (via dynamic service binding), but also its limitations, in particular that of relying on the assumption of guaranteed communication. In siCPS, this assumption is not always plausible, as entities need to operate even detached from their peers when deployed in ad-hoc infrastructures with no communication guarantees. This mismatch breaks the notion of autonomy at the implementation level and hinders the straightforward use of agent-based platforms in the development of siCPS.

With respect to AOSD methodologies, we notice that they point to the right direction of bridging the gap between the different design and development phases via a set of guidelines on how to transition between the phases. Tropos, in particular, provides a small and manageable set of concepts (actors, goals, plans) throughout the phases, from early requirements to implementation and deployment. On the down side, there is a small degree of automation in the mapping between the phases; in most case the transitions relies heavily on subjective design decisions requiring experience and ingenuity from the designers [YLL+08]. This is – in a sense – inevitable, as Tropos addresses a broad range of software systems (although it initially focused on Internet-based information systems). Nevertheless, in the development of large complex systems such as siCPS, automation (e.g. via model-driven techniques) is highly desirable, as it effectively reduces the development time and effort and helps manage the complexity therein.

The component-based paradigm also helps in reducing the development effort by providing reusable implementation artifacts (components) and utilities to compose them together. Contrary to the agent-based paradigm, CBSD emphasizes dependability (e.g. via model checking of assembled system configurations [HI10]) at the expense of adaptability and flexibility in interactions. That said, component-based approaches that provide some form of limited adaptability, typically in the form of mode switching [HKMU06], do exist.

Evaluating the overviewed component-based approaches, Kevoree focuses on the deployment of reusable software components to computational nodes. It features a rigid architecture structure that can be adapted at runtime and fits well domains where limited adaptation is required in response to environment dynamicity (e.g. smart spaces, where software components have to migrate from node to node to achieve load balancing). Communication between components in Kevoree relies on predefined connectors

(channels). Helena and DEECo, on the other side, feature the concept of ensembles to model groups of software components, dynamically created and disbanded via the runtime platform, that collaborate to achieve a common goal. In both cases, the architecture structure is not static, but "emerges" according to the specification of ensembles (which take over the role of traditional CBSD connectors). Communication is strictly bound to components that belong to the same ensemble. The communicating parties are also not referenced directly (e.g. via specifying the signatures of interacting components). Rather, the attributes of the communicating parties, i.e. the roles that components can undertake, determine the source and target of communication – a concept called *attributed-based communication*, featured by novel coordination languages (e.g. SCEL [DNFLP13]). This provides an effective way to deal with environment dynamicity and change at runtime, as changes are directly reflected in the architecture of the system, which is self-organized.

The differences between Helena and DEECo lie in their implementation mechanisms and their underlying assumptions. While Helena relies on an underlying messaging mechanism to bootstrap the creation of an ensemble and enable the managing and communication of components in an ensemble, DEECo features knowledge exchange as the only available communication style. While knowledge exchange can be also implemented via message passing, it does not have to; in fact, the Java implementation of DEECo performs knowledge exchange via directly transmitting data chunks, constituting the knowledge of each component, via wireless channels and reconstructing the knowledge on the receiving end [BGH+14a]. The observation is thus that Helena relies on stable network infrastructure, while DEECo considers temporary disconnections (typical in wireless ad-hoc networks) and related data inaccuracy as inherent properties of the underlying infrastructure.

Under this prism, we view DEECo as a comprehensive synergy of conceptual autonomy, featured in AOSD, and separation of concerns, featured in CBSD. In particular, DEECo abstractions and implementation mechanisms fit to the domain of siCPS, where high dynamicity in the underlying infrastructure (including the network infrastructure), physical distribution, and opportunistic communication are the norm. We thus choose the ensemble-based systems as featured in DEECo as our basis upon which we build the design methods that form the core of this thesis.

2.3 Goals Revisited

In the light of the state of the art discussed in the previous sections, we revisit here the research goals set out in Section 1.3 and provide a justification for the research performed in the context of this thesis in order to meet them.

The domain we are focusing on is that of software-intensive Cyber-Physical Systems (siCPS), i.e. distributed and highly dynamic cyber-physical systems of which software is the primary constituent. The design-time problem we are dealing with can be summa-

rized in the question: "How to design applications deployed on siCPS so that the applications' high-level goals can be consistently mapped to implementation-level artifacts?" The runtime problem we are dealing with can be summarized in the question: "How to trace the runtime behavior of applications deployed on siCPS to high-level goals in order to achieve runtime compliance checking?"

To answer these questions, we are looking into the specifics of the siCPS domain, namely the physical distribution of nodes, environmental uncertainty, high dynamicity, etc. To cope with the dynamicity of siCPS at the middleware level, we intend to rely on the concept of ensemble-based component systems featured by DEECo component model (Section 2.2.2.3). To come up with a systematic design method for siCPS, we intend to apply principles from goal-oriented requirements engineering (Section 2.1.1) and embed them to the development life cycle in a similar way as in agent-based software development methodologies, in particular Tropos (Section 2.2.1.1).

Specifically, our original research goals G1 and G2 (set out in Section 1.3) are refined into the following concrete objectives:

O1 Identify the specifics of siCPS that prevent the use of contemporary design approaches (including goal-oriented ones), and specify how principles from goal-oriented design approaches can be employed to deal with these specifics.

O2 Propose a design process for siCPS consisting of both a dedicated method and associated models. The method should bridge the gap between requirements analysis and architecture design and provide traceability between the high-level goals of a siCPS application and the system activities that maintain its operational normalcy.

O3 Extend the proposed method and associated models to allow capturing variability in the design of a siCPS application. The extended method should support architecture-based self-adaptivity in the form of choosing the system activities that collectively satisfy the high-level goals of the application, at each distinct situation the siCPS might reside in.

O4 Evaluate the proposed design method by mapping the associated design models to implementation-level abstractions featured by ensemble-based component systems. The feasibility of the method should be demonstrated by (i) realizing such a mapping based on existing ensemble-based component systems, and (ii) by performing empirical studies with software architects. The applicability of the method should be demonstrated by embedding it to a development methodology for ensemble-based systems.

With respect to the original goals, O1 to O3 correspond to G1, while O4 corresponds to G2.

Chapter 3

Overview of Contribution

This chapter highlights the key results of the thesis and then provides a detailed overview of the contribution.

In summary, key results of this thesis include:

- An analysis of the siCPS domain and the problems it poses to contemporary software engineering practices [GKB+14] and requirements-oriented design methods [GBH13];
- A novel method for designing applications for siCPS that supports both dependability [BGK+15, KBP+13] and self-adaptivity aspects [GBH+15b, Ger14];
- An assessment of the feasibility and applicability of the proposed method via (i) embedding it to a development methodology for siCPS [BDNG+13], (ii) implementation, integration, and experiments in an ensemble-based component system [GBH+15b], and (iii) performing an empirical study [GBH+15b].

The concrete objectives **O1**-**O4**, as set out in Section 2.3, are addressed in the following way.

Regarding **O1**, we identified the specifics of CPS that create the need for a different approach in their design and development [GKB+14]. We pinpointed several aspects, such as reachability and availability of global state, that are typically assumed in the software development processes of general-purpose systems (e.g. Internet-based systems), but are violated in the domain of siCPS. This hinders the application of ready-made methods and techniques in the design and development of siCPS. Our main observation is that time and physical distribution are primary concerns in siCPS and often neglected in contemporary design and development methods for large distributed dynamic systems. For example, agent-based methodologies (e.g. Tropos [BPG+04], Gaia [WJK00], O-MaSE [DeL14]) do not account for the impact of delays (caused, e.g., by unreliable communication) in agent interactions. Component-based approaches (based on popular component models such as CCM [WSO01] or OSGi [HPMS11]) do not provide support for reflecting the changes in the physical world (e.g. components moving in the environment and getting disconnected from their peers) to the level of system architecture. As a remedy, we advocated a novel synergy based on concepts from agent-based,

ensemble-based, component-based, and real-time and control systems centered on DEECo, an ensemble-based component system (Section 2.2.2.3). In parallel, we overviewed contemporary approaches that focus on goal-oriented requirements engineering and allow for systematic gradual design, and assessed their applicability in the domain of siCPS [GBH13]. Although elements from these approaches provided useful inspiration towards early validation, e.g. threat modeling in KAOS, and design of open-ended systems with multiple stakeholders, e.g. roles in Tropos, the conclusion was that they could not be applied as-is in the software engineering of siCPS. The main problem is that, by trying to be generally applicable, they do not provide an intuitive way to model repeatable system activities in siCPS that need to adhere to certain temporal constraints, and do not provide adequate support for connecting requirements to system design. In return, we sketched a novel method based on the iterative refinement of predicates that capture the repeatable activities (operational normalcy) of the system.

Regarding **O2**, we refined the predicate concept proposed in [GBH13] into the "invariant" concept. On this basis, we proposed a novel, requirements-driven, formally grounded method for the design of siCPS that focuses on the dependability aspect – the Invariant Refinement Method (IRM) [KBP+13]. In IRM, high-level goals and low-level system obligations (specifications of system activities) are captured as invariants over the knowledge of components in the system. Our interpretation of invariants is not that they should hold always, but frequently enough. Invariants are iteratively decomposed to the level that they can be mapped to obligations of individual components or to obligations of groups of components (ensembles). The result of the decomposition is an IRM model, canonically represented as a tree. This model is used in the next steps of the design process, which involve a straightforward mapping of the leaves of the tree to component processes and ensembles complying to DEECo. In order to assist the designer and provide correct-by-construction guarantees, we proposed [KBP+13] and later formalized [BGK+15] a set of invariant patterns for IRM. The patterns have the dual role of (i) providing a base for formal reasoning on an abstraction level higher than the level of individual knowledge valuations; (ii) helping a designer perform a correct refinement of invariants (via decomposing them) by leveraging on the relations between the patterns [BGK+15]. The value of IRM lies in its intuitive and manageable set of concepts (invariants, components, component knowledge) and in the straightforward connection it provides between requirements and system design that follows the architecture paradigm of components forming dynamic collaboration groups. The novelty lies in that it applies decomposition of requirements based on the temporal constraints associated with each requirement. To the best of our knowledge, IRM is also currently the only design method that is tailored to ensemble-based component systems.

Regarding **O3**, we extended our previous work on IRM to model alternative decompositions in the IRM model [GBH+15b, Ger14]. This allows modeling alternative realizations of system requirements – an essential step for a siCPS to change its behavior in the context of different runtime situations (system states). The extension, called IRM for self-adaptation (IRM-SA), allows for capturing the design alternatives and applicable configurations along with their corresponding situations. Design alternatives in IRM-SA are

modeled as alternative sub-trees in the IRM-SA model, while situations are modeled via one or more assumptions, i.e. invariants that should hold in the environment of the system. Self-adaptation in IRM-SA takes the form of architecture reconfiguration performed via three recurrent steps: (i) determining the current situation, (ii) selecting one of the applicable configurations, and (iii) reconfiguring the architecture towards the selected configuration. One of the novelties of IRM-SA is the ability to deal with operational uncertainty (e.g. temporary disconnections, hardware malfunctions) by allowing reasoning on the inaccuracies of the belief of individual components of an ensemble-based component system. This ability is absent in other approaches used in specifying self-adaptive behavior, such as dynamic software product lines [HPS12]).

Regarding **O4**, we mapped the IRM-SA concepts to DEECo concepts. The resulting systematic model-driven process starts with requirements elaboration and architecture design and ends with the specification of DEECo components, component processes and ensembles [GBH+15b, Ger14]. In particular, the leaves of the IRM-SA model are translated to DEECo component processes and ensembles via a systematic process that involves elaborating each individual leaf invariant and corresponds to the detailed design phase that typically follows architecture design. To support the design task and provide early structural validation and code generation, we created a GMF-based prototype of an IRM-SA editor (Section 5.2.1). To support quick prototyping and experimentation, we implemented an EMF-based IRM-SA self-adaptation mechanism and embedded it as a plugin to jDEECo (Section 5.2.2). This allowed us to experiment with the application of IRM-SA self-adaptation in decentralized settings (Section 5.2.2.1). The feasibility and effectiveness of the IRM-SA design process was evaluated by a controlled experiment with students (Section 5.3). Finally, the applicability of the proposed approach was demonstrated by positioning it in the ensemble development life cycle in [BDNG+13], a novel methodology that emerged within the ASCENS FP7 project [1] for the development of ensemble-based systems.

Chapter 4

Commented Collection of Papers

The main contributions of this thesis were published separately in various international conference and workshop proceedings. This chapter includes both summaries and full versions of the selected papers in the order presented in Section 1.4, as well as comments on the workshops and conferences where the papers were presented.

4.1 Software Engineering for Software-Intensive Cyber-Physical Systems

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

In proceedings of the **44th Annual Meeting of the German Informatics Society – 44. Gesellschaft für Informatik Jahrestagung (INFORMATIK 2014)**.

Published by Gesellschaft für Informatik, Bohn
pages 1179-1190,
ISBN 978-3-88579-626-8,
September 2014.

The original version is available electronically from the publisher's site at http://subs.emis.de/LNI/Proceedings/Proceedings232/article73.html .

Summary of the Paper

This paper, published as [GKB+14], serves as an introduction to the domain of software-intensive Cyber-Physical Systems (siCPS) and sets the context for the rest of the work presented in this thesis. In the paper, siCPS are defined as "a class of CPS that are software-intensive and, at the same time, distributed at a large scale, inherently dynamic, self-adaptive, self-aware, exhibiting emergent behavior, and safety-critical" – a definition that corresponds well to the working definition of siCPS provided in the introduction of this thesis (Section 1.1). Although the paper does not focus specifically on architecture design of siCPS, it provides useful insight on the problems that can arise in different software engineering (SE) phases of siCPS, including the architecture design phase, therefore contributing to the fulfillment of objective **O1**.

The main idea of the paper is to suggest that traditional SE models and methods are not sufficient in the design and development of siCPS, and give an argumentative explanation as to why this holds. The main line of argumentation is that traditional SE of general-purpose software systems (GPSS) relies on a number of key assumptions used to simplify the software development, such as static physical structure, location obliviousness, clique connectivity, and others. In siCPS, however, most (or even all) of these assumptions are not plausible. This renders traditional SE models and methods inapplicable to the domain of siCPS and calls for new SE abstractions, models and methods that reflect the specifics of siCPS. Apart from the assumptions that are violated in siCPS, these specifics also include a number of opportunities that can be advantageously exploited when developing siCPS.

Contemporary approaches that reflect the siCPS specifics, such as agent- and ensemble-based systems, are then analyzed in terms of their applicability in the domain of siCPS. The outcome of the analysis is that each approach deals with the challenges only partially and that a synergy of them is needed in a holistic framework. As a particular example of such a synergy, the DEECo component system (Section 2.2.2.3) is put forward. DEECo provides the necessary abstractions for software encapsulation and composition, and adequate runtime support to deal with most siCPS specifics. However, using DEECo has a number of implications that can lead to rethinking the development process of siCPS. Some of these implications, e.g. viewing DEECo components as autonomous agents and the decentralized DEECo-based operation, are discussed last.

Comments on Authorship

My personal contribution to this paper lies in analyzing the approaches that partially reflect the specifics of siCPS (agent-based systems, ensemble-based systems, real-time and control systems, and MANET and gossip protocols). Based on this, and under helpful guidance and supervision of the other authors, I compiled a list of SE specifics for siCPS comprising the traditional SE assumptions of general-purpose ICT systems that are violated in siCPS and the opportunities that can be exploited in siCPS. Moreover, again under helpful guidance and supervision of the other authors, I authored a majority of the text.

Software Engineering for Software-Intensive Cyber-Physical Systems

Ilias Gerostathopoulos, Jaroslav Keznikl, Tomas Bures, Michal Kit, Frantisek Plasil

Faculty of Mathematics and Physics
Charles University in Prague
Malostranske Namesti 25
11800 Prague, Czech Republic
{iliasg, keznikl, bures, kit, plasil}@d3s.mff.cuni.cz

Abstract: In software-intensive cyber-physical systems (siCPS) the interplay of software control with the physical environment has a prominent role. Nowadays, siCPS are expected to (i) effectively deal with the issues of distribution, scalability, and environment dynamicity, (ii) control their emergent behavior, and, at the same time, (iii) be versatile and tolerant in face of changes and threats. Although approaches that individually meet the above requirements of siCPS already exist, their synergy in a comprehensive software engineering framework is far from trivial. In this paper, we pinpoint the important characteristics of engineering siCPS in an attempt to show that they introduce distinct challenges to traditional software engineering. We argue that this can be addressed by a synergy and adaptation of existing models and abstractions, show our proposal towards such a synergy, and discuss its implications.

1 Introduction

Cyber-physical systems (CPS) are systems of collaborating elements which closely interact with their environment by sensing and actuating. Typically, CPS are characterized by being decentralized, distributed, and heterogeneous.

With the proliferation of smart embedded and mobile devices (smart phones, intelligent cars, etc.) and wireless networks, there is a further trend of CPS becoming large-scale pervasive systems, which combine data from various sources to control real-world ecosystems (e.g., intelligent traffic control, which gathers data about traffic from cars and other sensors in a city and uses them to navigate cars, control the traffic lights, and manage parking allocation). An important feature of these systems is that they are adaptive in order to adjust to situations in the physical environment, and they exhibit emergent behavior (i.e., behavior that comes about as the joint product of behaviors and interactions of many elements of the system). These CPS are also highly dependent on software – they are software-intensive systems [HRW08]. This means that software is by far the most important and most complex constituent of modern CPS.

Continuous dependable operation of CPS is particularly important as the close connection to the physical environment frequently renders the functionality of CPS safety-critical (e.g., operation of the traffic lights in the intelligent traffic control). In addition to being dependable, the software of CPS has to be able to adapt to changing situations in the physical environment. Ideally, it should possess some self-awareness and self-healing properties to cope with not fully anticipated situations.

Along the lines above, in this paper we consider a class of CPS that are software-intensive and, at the same time, distributed at a large scale, inherently dynamic, self-adaptive, self-aware, exhibiting emergent behavior, and safety-critical. It is also important to note that these CPS are targeted by on going research agendas (e.g., EU framework Horizon 2020). We will refer to these CPS as *software-intensive CPS* – siCPS. We argue that siCPS have a number of specifics, which prevent to fully employ traditional software models and software engineering methods. This calls for tailored models and software engineering abstractions that address and potentially take advantage of the specifics of siCPS [Le08]. In fact, siCPS reach the threshold when it is disputable whether we are still dealing with tailored traditional software engineering or whether we are encountering a new paradigm in computing.

As the particular contribution of this paper, (i) we overview these specifics (Section 2) and analyze how they can be addressed by a synergy and adaptation of existing software models and software engineering abstractions (Section 3). On the basis of this, (ii) we give a practical example of such a synergy (Section 4), namely DEECo, an ensemble-based component system [Bu13]. Finally, based on the lessons learned with DEECo, (iii) we discuss potential challenges stemming from the interplay of the models and abstractions in such a synergy (Section 5).

2 Software Engineering Specifics of siCPS

The large-scale physical distribution and interconnectedness within the physical environment makes siCPS rather specific in terms of software engineering (SE). In this section, we overview these specifics from the perspective of SE assumptions and opportunities.

2.1 SE Assumptions Violated in siCPS

A number of assumptions that are typically presumed in traditional SE of general-purpose software systems (GPSS) are violated in siCPS. The assumptions build on the fact that a lot of complexity related to networking and the environment can be considered low-level in GPSS and abstracted away by the operating system and middleware. Of course, even in traditional SE some key assumptions may be violated when developing GPSS with special needs (e.g., high-availability, open-endedness). Nevertheless, siCPS stand out by the large number of such violated assumptions.

Therefore, below we identify and discuss a number of assumptions in traditional SE of GPSS that we deem to have a significant simplifying effect on software development but – according to our experience – cannot be preserved in engineering siCPS:

A1 *Static physical structure* – Even though data and code are subject to mobility in GPSS, the physical nodes where the code is running are typically stationary. In siCPS, the physical substratum is continuously evolving, as nodes move in the physical environment. The fundamental challenge is how to map the ever-changing substratum to the network of computational nodes so that stringent requirements on the desired services are always met.

A2 *Location obliviousness* – The cost and profit of reaching a particular node is typically not significantly influenced by its physical location. This independence facilitates the creation of open-ended and dynamic distributed GPSS and is generally considered an asset. In siCPS, locality of peer nodes is a fundamental design constraint, since physical proximity directly affects reachability and connectivity on one hand and functional correctness on the other.

A3 *Reachability (clique connectivity)* – GPSS typically rely on the Internet network stack for the underlying communication protocols (Internet-based systems [Fr07]). This means that with high probability any node can successfully establish point-to-point communication links with any other node in the system. In siCPS there is no such guarantee, as nodes often operate over dynamic networks lacking a permanent infrastructure, such as mobile ad-hoc networks (MANETs). This limitation imposes a fundamental constraint in the design of siCPS, since nodes are expected to operate in full autonomy, even detached from their peers.

A4 *Stable connections* – In most GPSS, on top of being able to reach and connect to remote subsystems, connections are typically considered stable. This is manifested in the handling of communication errors in such systems: errors are considered exceptions and have to be handled accordingly. In siCPS, errors in communication are the rule, not the exception. Thus, they can no more be handled as exceptions. The property of unstable connectivity has to be acknowledged and ideally be reflected in the employed SE abstractions.

A5 *Availability of global state* – Reasoning over the global state of a distributed system is a requirement for many applications. Although techniques exist for traditional distributed GPSS (e.g., distributed consensus), they are not directly applicable to siCPS because of the loose connectivity among the nodes. Also, since the local state in siCPS evolves continuously with the physical environment, attaining global state is generally infeasible.

A6 *Marginality of real-time aspects* – GPSS typically do not impose hard real-time constraints on their operation and communication. When time matters (e.g., Internet-based video streaming applications), it is mostly because late responses may impede system performance rather than correctness. In siCPS, the passage of time becomes a central feature of system behavior and design, since stringent notion of time is fundamental for measuring, predicting and controlling properties of the physical environment.

A7 *Crisp consistency* – In traditional distributed GPSS, there is a crisp notion of data consistency – the data is either consistent or not (this includes also eventual consistency etc.). On the other hand, in siCPS, where strict distributed synchronization becomes too expensive, such interpretation of consistency is not desirable. Rather, in siCPS it is important to quantify and/or guarantee the degree of (in)consistency [Al14].

A8 *Controlled dynamism* – Many GPSS are dynamic in the sense that they dynamically adapt to changes and recover from malign states. This kind of dynamism, though, is typically a result of actions initiated by the system itself or its administrator. On the contrary, in siCPS, dynamism is inherent, imposed by the

physical environment itself. Thus, siCPS need to detect and recover from contingent and often unforeseen situations in their environment in a non-disruptive way and without supervision (they have to be self-aware and autonomic).

A9 *Focus on reactive behavior* – Outputs of a GPSS are typically reactions to explicit stimuli, such as service requests and internal/external events (e.g., computation is initiated as a response to user input). Instead of waiting for an event, siCPS have to operate proactively in order to react to and also perform changes based on properties that are either sensed or predicted. Relying on simple (e.g., rule-based) reaction patterns is insufficient, since it may lead to oscillations and instability.

A10 *Stateful communication* – GPSS usually assume stateful communication in the communication protocols they employ. This enables effective synchronization among distributed components. Moreover, since stable connections are assumed (A4), errors are treated as exceptional and detected and solved via explicit error recovery. In siCPS, stateful communication does not scale. In fact, extreme network dynamism, typical for siCPS, may incur recurrent error recovery.

2.2 SE Opportunities in siCPS

As pointed out in Section 2.1, none of the discussed assumptions can be generally presumed in siCPS. This makes it a non-trivial challenge to develop siCPS by applying traditional SE methods. However, it would be wrong to perceive all specifics of siCPS as impeding their development, since they may provide opportunities for getting around the violated assumptions. In this perspective, it is desirable to take advantage of such siCPS specifics instead of aiming at adapting traditional SE methods, e.g., building a complex middleware to provide a traditional programming model.

To pinpoint this idea, we have compiled a list of specifics, which we believe can be advantageously exploited in addressing the violated assumptions. Although not complete, we believe this list gives an important research direction for siCPS design methods:

O1 *Physical mobility* – Devices used in siCPS span from stationary to portable and mobile ones. Computational nodes deployed on mobile devices can carry information while moving. This contributes to the overall connectedness of the system, as a mobile node covers a much bigger physical area while moving, and can effectively spread the information in the area and connect otherwise disconnected network partitions. For example, a vehicle moving along a street segment can aggregate temperature data measured from sensors positioned in the tarmac along its route (which themselves cannot reach any external network), and publish the data on a remote server, or spread it to other vehicles in the vicinity.

O2 *Physical locality* – The fact that devices in siCPS are physically close provides a natural way to partition the system into subsystems based on geographical location. This is, again, special to siCPS; general-purpose systems are rarely partitioned based on physical location, because of the otherwise useful assumption on location obliviousness. Having such a natural partitioning can be easily exploited to achieve high levels of scalability.

O3 *Location-dependency of data* – Data in siCPS are often location-dependent, meaning that the value of certain measureable system attributes depend on the physical location of the sensors that provide the raw data. This dependency, in combination with the physical proximity of sensor nodes, allows for data sharing and reuse among nearby nodes and has the potential to contribute to system robustness (in face of sensor failures, etc.).

O4 *Physical laws in data evolution* – Since siCPS operation typically involves sensing physical-environment properties (e.g., position, battery capacity, temperature), one can take advantage of the physical laws that govern the evolution of the values of such properties to estimate/predict their real values. In effect, a value that is slightly stale can still be used, if certain safety bounds on its evolution in time can be established [Al14]. As an example, consider a wireless-based adaptive cruise control system: a stale value of the front vehicle's position can still be used by the rear vehicle's cruise control, since it is possible to estimate the actual position based on the maximum and minimum vehicle acceleration, typically provided by car manufacturers.

3 Approaches that Partially Reflect the Specifics of siCPS

There are no comprehensive methods or supporting models that address the specifics of siCPS in their entirety, as far as our research has indicated. Nevertheless, our experience shows that some SE approaches target these specifics at least partially. In this section we provide a short overview of such approaches (summarized in Table 1), with the goal to later show how they can be combined in a comprehensive framework.

Agent-based systems. In order to deal with dynamicity in siCPS, one can be inspired by *autonomous agents*. This abstraction brings conceptual autonomy to the loosely coupled system parts. Each part is designed to operate with a partial view of the whole system, beneficial when the global state is not available (A5). For example, in the Belief-Desire-Intention (BDI) architectural model [RG95], agents maintain a *belief* about the rest of the system to guide their autonomous decisions. In addition, *multi-agent systems* [SL08] feature the concepts of agent roles and groups, which bring the autonomy to architecture organization and allow building *self-organized systems* that do not rely on the assumptions of controlled dynamism (A8) and static physical structure (A1). An important problem is that industrial agent implementations do not translate the conceptual autonomy and the other useful agent notions (goals, intentions, roles, groups) into proper software engineering constructs that satisfy real-life requirements of autonomous behavior. In particular, they still rely on the assumption of relatively stable bindings between the agents (A4), which is not plausible in most siCPS.

Ensemble-based systems. Another important specific of siCPS is the opportunistic fashion of operation in a dynamic environment at a massive scale. To this end, the paradigm of *attribute-based communication* in *ensemble-based systems* has recently gained attention [De13]. Here, the target of communication is determined according to the values of its attributes rather than by a direct identifier. This paradigm can be exploited to model a best-effort, dynamic coordination of components, effectively dealing with cases when the assumptions of static physical structure (A1), reachability

Assumption:	Agent-based systems	Ensemble-based systems	MANET & gossip protocols	Real-time & control systems	DEECo
A1 Static physical structure		+	+		+
A2 Location obliviousness		+	+		+
A3 Reachability		+	+		+
A4 Stable connections	-		+	-	+
A5 Availability of global state	+	+			+
A6 Marginality of real-time aspects				+	(+)
A7 Crisp consistency		-		+	(+)
A8 Controlled dynamism	+	+	+	-	+
A9 Focus on reactive behavior			+	+	+
A10 Stateful communication				+	+

Table 1: Assumptions from Section 3 and DEECo: lifting "+", partially lifting "(+)", and specific reliance upon "-".

(A3), and controlled dynamism (A8) are violated. However, the application of this paradigm typically relies on explicit and crisp handling of data consistency (A7).

MANET and gossip protocols. At the network layer, extensive research in the areas of *mobile ad-hoc networks* (MANETs) has resulted into a number of routing protocols (see [NPD12] for a comprehensive review), which are able to operate over infrastructure-less dynamic networks. In MANETs, each node acts both as a host and as a router. Node mobility results in dynamically changing network topology. As such, MANET protocols lift the assumption of static physical structure (A1) and work even when the reachability assumption (A3) is violated, thus becoming very relevant to siCPS. Moreover, MANET protocols lift the assumption of location obliviousness (A2), as they enable position-based packet routing [MWH01] (sometimes called *geocast* routing). A promising synergy for siCPS is to combine geocast protocols at the network layer with gossip protocols at the data dissemination layer, effectively enabling proactive, opportunistic communication (A9) in MANETs [Fr07]. Integration of gossiping brings a remedy in cases of unstable connections (A4) and inherent dynamism (A8).

Real-time and control systems. As to strong interaction with physical environment, many techniques already exist in the domain of *embedded real-time systems* [Bu05] and *software control systems* [Pa12]. Such techniques promote proactive behavior (A9) and focus on real-time attributes (A6). They employ control feedback loops, which continuously maintain the operational normalcy (stability) of a system by adequate scheduling of periodic tasks. These techniques stand as a promising way to handle data outdatedness in absence of crisp consistency interpretation (A7) in siCPS, by effectively setting the bounds that define the range of normal system operation. Communication in embedded real-time systems is also typically stateless (A10); consider, e.g., data publishing on CAN bus. Nevertheless, real-time analysis and design typically rely on the assumption of predictable environment, which itself relies on controlled dynamism (A8) and stable connections (A4) assumptions.

4 DEECo: A Synergy

In order to evaluate the potential for a synergy of the approaches discussed in Section 3, as a particular example we present DEECo [Bu13, Ke12] – an *Ensemble Based Component System* that we have proposed specifically for architecting siCPS.

In DEECo, we take the approach of adopting component-based development (CBD) as the basic substratum on top of which we embed selected SE approaches from Section 3. CBD employs reuse, encapsulation and separation of concerns in order to manage the complexity of building and maintaining large applications [CL02]. In CBD, and thus also in DEECo, systems are built around well-defined architectures based on a composition of components, which themselves are seen as encapsulated, reusable, and substitutable entities.

In the remainder of this section, we describe the individual constituents of the DEECo component model with focus on how we approached the synergy. We refer the interested reader to [Bu13] for a detailed technical description of DEECo and for the formal semantics of DEECo. Also, a Java implementation is available[1].

4.1 Component

Adopting the ideas of agent-based and self-adaptive systems, the concept of *component* in DEECo is centered on the features of autonomy, self-adaptation, and belief (A5). Specifically, a component is an autonomous, encapsulated, and composable software entity constituting its own state and behavior.

As is typical for software agents, component state is expressed in terms of *knowledge* (e.g., line 3 in Figure 1). Note that in DEECo, all the data accessible to a component is referred to as knowledge. In alignment with the BDI architectural model, knowledge of a component comprises both the private component state (e.g., calendar) and the component's belief about the rest of the system (e.g., parkingAvailability). In slight difference from traditional BDI approach, rather than being updated explicitly by the component itself, the belief is updated automatically (by the execution environment, Section 4.3) as a result of component composition (Section 4.2). This decision further stresses the component's autonomy and separation of concerns.

The behavior of a component is represented by a set of *processes* (e.g., lines 4-7 in Figure 1). Following the notions of control systems and self-adaptive systems, a process is essentially a feedback loop, continuously and proactively maintaining the operational normalcy of a component (A9). At the same time, each process executes concurrently, independently of the other processes, i.e. it atomically reads its inputs, executes its body, and atomically writes its outputs. A process operates strictly upon the knowledge of the corresponding component; it may thus interact with other components only through the (externally updated) belief (A4, as there is no "direct" communication among components).

[1] https://github.com/d3scomp/JDEECo

```
1.  component Vehicle
2.    knowledge:
3.      calendar, parkingAvailability, plan, ...
4.    process computePlan(in calendar, in parkingAvailability, out plan):
5.      function:
6.        plan ← JourneyPlanner.computePlan(calendar, parkingAvailability)
7.      scheduling: periodic( 5000ms )
8.    ...
9.
10. component ParkingLot
11.   knowledge:
12.     position, availability, ...
13.   process monitorAvailability(out availability):
14.     …
15.
16. // updates Vehicle's belief about availability of all ParkingLots along the route
17. ensemble UpdateAvailabilityInformation:
18.   coordinator: Vehicle
19.   member: ParkingLot
20.   membership:
21.     ∃ event ∈ coordinator.calendar: distance(member.position, event.position) < TRESHOLD
22.   knowledge exchange:
23.     coordinator.parkingAvailability ← members.reduce(member.availability)
24.   scheduling: periodic( 2000ms )
```

Figure 1: Example of a DEECo component and ensemble definition in a DSL.

4.2 Component Composition

For component composition we adopt the approach of ensemble-based systems and multi-agent systems by employing autonomic self-organization of components into component *ensembles* (in multi-agent systems called *groups*). This self-organization is based on a declarative representation of a component's membership in an ensemble, based on the component's context (A1 and A3). In order to distinguish in which ensemble the membership is being decided upon, every ensemble has a *coordinator*. Membership in an ensemble with a given coordinator is based on whether a component is able to assume the role of a *member* w.r.t. the coordinator. This is expressed technically via a *membership condition*, which decides whether two given components can form a coordinator-member pair. Following the idea of attribute-based communication, the membership condition is defined upon the attributes (i.e., knowledge exposed for this purpose) of the components in question (e.g., line 21 in Figure 1). Note, that the ensemble definition is generic and determines ensemble instantiation for each group of components meeting the membership condition (w.r.t. particular coordinator). Also, a component can be a member or coordinator of multiple ensembles at the same time. Within an ensemble, we adopt the idea of stateless, proactive communication employed in control systems and gossip-based systems (A9 and A10). Specifically, the communication takes the form of stateless *knowledge exchange*. Its objective is to update the belief of the components within the ensemble recurrently and proactively, based on a given prescription (e.g., line 23 in Figure 1). This form of communication aligns well with the proactive, cyclic execution of component processes. Note, that the statelessness and proactivity make knowledge exchange suitable for cases of faulty connections (A4) and inherent dynamism (A8).

4.3 Execution Environment

The main task of the DEECo execution environment is performing knowledge exchange in a distributed setting. For this purpose, we combine the protocols for geographical routing in MANETs with gossip protocols so as to enable location-aware communication of belief (A2) in mobile ad-hoc environments (A1 and A3) with unstable connections and inherent dynamism (A4 and A8). Specifically, the execution environment proactively advertises the knowledge of a (source) component to all the other potentially-interested (target) components via a geocast protocol. Then, in case the source and target components meet the membership condition of an ensemble, the execution environment updates the belief of the target component according to the knowledge exchange prescription of the ensemble.

Adopting the approach of embedded real-time systems, the execution environment is also responsible for execution of component processes and activities related to knowledge exchange in a (soft) real-time fashion (A6 and partially A7), featuring both periodic and event-based scheduling.

5 Discussion of Implications

Engineering siCPS with the basic building blocks (autonomous components, ensembles) offered by the proposed synergy in DEECo offers several advantages, but also poses new challenges. As seen in Table 1, DEECo addresses all of the identified challenges of A1-A10, which we deem a step forward. Certainly, there could be other assumptions, e.g., predictability of underlying platform and global synchronization of beliefs, which still remain to be addressed. Building on our experience in applying the ensemble-based component system paradigm to two real-life case studies, namely the intelligent vehicle navigation [Bu13] and the firefighter tactical coordination [Bu14], this section discusses the implications of merging different methods.

Exploitation of the opportunities. A close synergy of geocast MANET protocols and attribute-based communication, and an integration of membership evaluation and routing in particular, allows exploiting new opportunities based on physical locality (*O2*) and location-dependency of data (*O3*) (i.e., membership can effectively exploit physical location). Further, the proactive gossip-based advertisement of belief enables exploiting the physical mobility (*O1*). The cyclic and real-time nature of component processes also facilitates use of models that estimate/predict the safety bounds of knowledge evolution [Al14]. This is done by exploiting the physical laws that govern the evolution of certain knowledge values (*O4*).

Components as autonomous agents. Borrowing the ideas of belief and autonomous operation from agent-based systems and coupling them with the encapsulation and deployment facilities of component-based systems results into a dependable platform for robust component-based agent implementations. The robustness is achieved by grafting such “agents” with implicit component binding and communication. Contrary to other agent-based frameworks, the autonomous components thus do not communicate directly, e.g., via sending messages; instead, component knowledge serves as a communication medium. A component’s belief, i.e., the part of its knowledge that reflects knowledge of

other components, plays a role of "smart" sensors and actuators. For instance, a belief could represent a "smart" sensor providing "positions of up to 10 closest parking lots, which are available". All in all, a component's belief is updated externally – via knowledge exchange handled by execution environment.

Stateless interaction. Adopting the idea of attribute-based communication in component interaction has many advantages when considering that components in siCPS recurrently appear and disappear and form dynamic groups of best-effort coordination. At the same time, having no means of direct component binding and addressing makes it challenging – but certainly not impossible, as we have observed – to realize some forms of protocol-based communication. This is essential in certain interactions, e.g., reserving of a parking place by a specific vehicle at a specific parking lot. Stateless interaction dictates knowledge design in a way that it is always possible to reconstruct the state of the session from the knowledge, e.g. by assigning each parking reservation request a globally unique identifier (GUID), so that a reservation response could refer to it.

Embedded feedback loops. When designing siCPS, special means have to be provided for feedback loops. By building on the ideas of control and real-time systems, DEECo embeds the feedback loop operation both at design time and runtime. Systems based on feedback loops typically require a description of operational normalcy, usually in terms of periodic scheduling of tasks. However, the adoption of this idea needs a paradigm change in the design process, to explicitly focus on the normalcy that each process is expected to maintain as opposed to goals to be achieved [Ke13].

Decentralized operation. Coupling best-effort data dissemination of MANET protocols with attribute-based communication and decentralized system operation can result in situations when different parties act based on inconsistent local beliefs – so-called split-brain scenarios. For instance, a component can believe itself to be a member of an ensemble, while the ensemble's coordinator does not recognize this situation (or vice-versa). This behavior is in a way inevitable, however it has to be accounted for in the design, e.g., by making components only weakly synchronized or by relying on an underlying network or physical environment to provide some guarantees (thus making these split-brain situations temporary with an upper bound for duration).

Ensembles as component connectors. The duality between components and ensembles resembles the classical problem of components and connectors – especially whether connectors are only special types of components and what functionality should be in connectors and what functionality should be in components. In particular, this holds when connectors comprise complex adaptation logic. In DEECo though, this problem is partially remedied by distinguishing that (i) although stateful, a component has a direct access solely to its local knowledge, (ii) an ensemble embodies only stateless exchange of knowledge among its member components. This is a strong conceptual difference pushing ensembles into the role of simple connectors and components into the role of entities performing the actual computation and data aggregation.

Parallel process execution. The physical world is inherently concurrent. Software engineering abstractions for engineering siCPS have to deal with concurrency by allowing execution of processes in parallel. This leads to challenges with regard to the handling of shared resources, which, if not dealt with, can result into race conditions,

deadlocks, etc., effectively jeopardizing the safety of the system. Similar to actor-based design, where the exchanged data are considered immutable, DEECo avoids introducing any dedicated synchronization constructs. Rather, it employs the simple semantics of atomically operating over knowledge while applying the rule of single-writer for each knowledge field. The downside of the approach is that it sometimes leads to the necessity of having a special "aggregation" process in a component, which merges data coming from different sources (similar situation happens in actor-based approaches as well). However, this seems a reasonable price to pay for preventing race conditions by design.

6 Related Work

Since CPS is an emerging class of systems, there are multiple research efforts trying to shed light on the state of the art and the challenges ahead [KK12, Sh09]. Unfor- tunately, not as many solutions are proposed, especially when considering guidance via proper software engineering abstractions specific to CPS. Our work highlights the problems in CPS software engineering, while, at the same time, we propose solutions to these problems and evaluate their implications. In the same spirit, in [DLS12], Derler, Lee and Vincentelli focus on the challenges with modeling CPS caused by the intrinsic heterogeneity, concurrency, and sensitivity of such systems. Backed up by a hybrid-system-modeling environment called Ptolemy II, their approach emphasizes determinism and predictability in modeling and simulations of safety-critical CPS. In [Le08], Lee reviews the requirements/specifics of CPS and identifies the absence of timing behavior in core abstractions in computing as the main impediment in developing future CPS. In our work, we focus on the subset of CPS that is software-intensive, where structural models and systematic engineering methods become more relevant.

Our aim at a synergy can be compared to frameworks proposed for self-adaptive/self-organizing systems, e.g., [DFR10], and autonomic agent-based systems, e.g., [LPH04]. In [DFR10], Di Marzo Serugendo, Fitzgerald and Romanovsky propose a synergy of self-organization, agent-inspired autonomy and rule-based reasoning into a service-oriented architectural framework. Their approach is centered around the concepts of self-describing components, component metadata and interaction policies executed at runtime, resembling the concepts of components, component knowledge and ensembles, respectively. In [LPH04], Liu, Parashar and Hariri present a component-based framework for autonomic agents building on agent-based middleware infrastructure. The difference from these and other similar approaches lies in the fact that we deal with the specifics of siCPS, where unreliable communication and extreme dynamism loom large.

7 Conclusion

Building software for software-intensive cyber-physical systems (siCPS) is far from trivial. In this paper, we attempted to pinpoint the challenges and pitfalls associated with applying traditional software engineering (SE) methods in siCPS and to show how these challenges can be met by a comprehensive synergy and adaptation of existing SE models, methods and abstractions. This we exemplified on the DEECo component model. The evaluation of the proposed synergy in DEECo outlines a number of

interesting research topics in terms of addressed and waiting-to-be-addressed issues, such as design based on maintaining operational normalcy.

References

[Al14] Al Ali, R. et al.: Architecture Adaptation Based on Belief Inaccuracy Estimation. In: Proc. WICSA'14, Sydney, Australia, 2014. IEEE, 2014; pp. 87-90.

[Bu05] Buttazzo, G. et al.: Soft Real-Time Systems: Predictability vs. Efficiency. Springer, 2005.

[Bu13] Bures, T. et al.: DEECo: An Ensemble-based Component System. In: Proc. CBSE'13, Vancouver, Canada, 2013. ACM, 2013; pp. 81-90.

[Bu14] Bures, T. et al.: Adaptation in Cyber-Physical Systems: from System Goals to Architecture Configurations. Tech. Rep. D3S-TR-2014-01, Charles University.

[CL02] Crnkovic, I.; Larsson, M.: Building Reliable Component-Based Software Systems. Artech House, Inc., Norwood, MA, USA, 2002.

[De13] De Nicola, R. et al.: A Language- Based Approach to Autonomic Computing. In: Formal Methods for Components and Objects, Springer, 2013; pp. 25-48.

[DFR10] Di Marzo Serugendo, G.; Fitzgerald, J.; Romanovsky, A.: Meta-Self: An Architecture and a Development Method for Dependable Self-* Systems. In: Proc. 25th ACM Symp. on Applied Computing, Sierre, Switzerland, 2010. ACM, 2010; pp. 457–461.

[DLS12] Derler, P.; Lee, E. A.; Sangiovanni-Vincentelli, A.: Modeling Cyber-Physical Systems. In: Proceedings of the IEEE, 100(1):13-28, Jan 2012.

[Fr07] Friedman, R. et al.: Gossiping on MANETs: The Beauty and the Beast. In: ACM SIGOPS Operating Systems Review, 41:67–74, 2007.

[HRW08] Hölzl, M.; Rauschmayer, A.; Wirsing, M.: Software-Intensive Systems and New Computing Paradigms. In: Engineering of Software-Intensive Systems: State of the Art and Research Challenges, Springer-Verlag, 2008; pp. 1-44.

[Ke12] Keznikl, J. et al.: Towards Dependable Emergent Ensembles of Components: The DEECo Component Model. In: Proc. of WICSA/ECSA'12. IEEE, 2012; pp. 249–252.

[Ke13] Keznikl, J. et al.: Design of Ensemble-based Component Systems by Invariant Refinement. In: Proc. of CBSE'13, Vancouver, Canada, 2013. ACM, 2013;pp. 91-100.

[KK12] Kim, K.-D.; Kumar, P.R.: Cyber-Physical Systems: A Perspective at the Centennial. In: Proceedings of the IEEE, 100 (Special Centennial):1287-1308, May 2012.

[Le08] Lee, E. A.: Cyber Physical Systems: Design Challenges. In: Proc. ISORC'08, Orlando, Florida, USA, 2008. IEEE, 2008; pp. 363–369.

[LPH04] Liu, H.; Parashar, M.; Hariri, S.: A Component-Based Programming Model for Autonomic Applications. In: Proc. ICAC'04, New York, USA, 2004. IEEE; pp. 10-17.

[MWH01] Mauve, M.; Widmer, A.; Hartenstein, H.: A Survey on Position-based Routing in Mobile Ad Hoc Networks. In: IEEE Network, 15(6):30-39, Nov 2001.

[NPD12] Natesapillai, K.; Palanisamy, V.; Duraiswamy, K.: A Review of Broadcasting Methods for Mobile Ad Hoc Network. In: International Journal of Advanced Computer Engineering, Serial Publications, New Dehli India, Sep 2012.

[Pa12] Patikirikorala, T. et al.: A Systematic Survey on the Design of Self-Adaptive Software Systems using Control Engineering Approaches. In: 2012 ICSE Workshop on Soft. Eng. for Self-Adaptive and Self-Managing Systems, Jun 2012. ACM; pp. 33–42

[RG95] Rao, A. S.; Georgeff, M. P.: BDI Agents: From Theory to Practice. In: Proc. of the 1st Int. Conf. on Multi-Agent Systems, 1995; pp. 312–319.

[Sh09] Sha, L. et al.: Cyber-Physical Systems: A New Frontier. In: Machine Learning in Cyber Trust, Springer US, 2009; pp. 3–13.

[SL08] Shoham, Y.; Leyton-Brown, K.: Multiagent Systems: Algorithmic, Game- Theoretic, and Logical Foundations. Cambridge University Press, 2008.

4.2 Position Paper: Towards a Requirements-Driven Design of Ensemble-Based Component Systems

Ilias Gerostathopoulos,
Tomáš Bureš,
Petr Hnětynka

In proceedings of the **2013 International Workshop on Hot Topics in Cloud Services (HotTopiCS 2013)**.

Published by ACM,
pages 79-86,
ISBN 978-1-4503-2051-1,
April 2013.

The original version is available electronically from the publisher's site at http://dx.doi.org/10.1145/2462307.2462325.

Summary of the Paper

This paper, published as [GBH13], positions the work leading to this thesis in the context of requirements-driven design of siCPS. In the paper, the term *ensemble-based component systems* (EBCS) is used to refer to siCPS modeled according to the ensemble paradigm, as proposed by component-based approaches such as Helena (Section 2.2.2.2) and DEECo (Section 2.2.2.3). The term "requirements-driven design" refers to architecture design that starts from the requirements elaboration phase and continues in a systematic way to the next phases, i.e. the architecture and detailed design phases.

The main idea of the paper is to identify and elaborate on the characteristics of EBCS (i.e. the notion of belief and its explicit management, the isolated component computation, and the dynamic component bindings), the challenges that they pose in a systematic requirements-driven design of EBCS, and the promising methods and abstractions that can be employed in such a design. As such, it directly addresses the objective **O1**. Having identified goal-oriented requirements engineering (GORE – Section 2.1.1) as an area worth exploring, two contemporary approaches are described and evaluated in terms of their applicability in the requirements-driven design of EBCS. The outcome is that both KAOS (Section 2.1.1.1) and *i** (Section 2.1.1.3) combined with Tropos (Section 2.2.1.1) have their limitations. In particular, KAOS provides little guidance into mapping a set of requirements to architecture artifacts, and *i**/Tropos fail to address the challenges related to emergent architectures and real-time computation needs in EBCS. Nevertheless, they both point to the right direction of focusing on system-level goals that provide a high-level view over the nonvolatile information that needs to be captured and elaborated in a systematic design process.

The inspiration drawn from the overviewed GORE approaches, together with the analysis of the EBCS specifics, leads to proposing a novel design method tailored to the domain of EBCS – the Predicate Refinement Method (PRM). PRM is the precursor of the Invariant Refinement Method (Section 4.3). The main idea of PRM is modeling requirements as predicates over the knowledge of the stakeholders (components in a DEECo-based system). Such predicates become *invariants* in the idealized case where computation and communication are instantaneous; in a real-word system, however, they should hold "frequently enough". Although PRM is presented in this paper as a method that complements DEECo abstractions at the design level, it can be generalized to any EBCS.

Comments on Authorship

My personal contribution in this paper lies in surveying the state of the art in goal-oriented requirements engineering and identifying the two representatives – KAOS and i*/Tropos – and critically evaluating their applicability, their strengths and weaknesses in the domain of siCPS. I also came up with the running example of intelligent vehicle navigation, and elaborated the idea behind the Predicate Refinement Method, which originally stemmed from the collaboration with my supervisors (co-authors in this paper). Finally, under the indispensable guidance of my supervisors, I authored a majority of the text.

Position Paper: Towards a Requirements-Driven Design of Ensemble-Based Component Systems

Ilias Gerostathopoulos, Tomas Bures and Petr Hnetynka
Faculty of Mathematics and Physics
Charles University in Prague
Malonstranske namesti 25, Prague, Czech Republic
iliasg,bures,hnetynka@d3s.mff.cuni.cz

ABSTRACT

Although approaches that effectively address the distribution and dynamism of adaptive systems at a middleware level exist, the design of complex, ensemble-based systems still remains a significant challenge. This hinders the development of real-life applications based on the ensemble paradigm. A promising approach appears to be the coupling of proven low-level concepts with high-level ones, revisiting requirements modeling in the realm of ensemble-based systems. To this end, the goal of this paper is to point out the specific challenges related to the design of ensemble-based systems and show that classic requirements models and methods cannot be applied out-of-the-box in a requirements-driven design of ensemble-based applications. In response to this problem, a novel design method based on the iterative refinement of system requirements expressed by predicates on stakeholder's knowledge is discussed.

Keywords

Ensemble based, requirements engineering, system design

Categories and Subject Descriptors

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

1. INTRODUCTION

Current trends in the area of information technology focus on the development of highly distributed systems comprised of sophisticated on-demand computing services, often characterized as *cloud systems*. A subset of such systems is the ones operating in the so-called *ad-hoc clouds*, i.e., in highly dynamic environments (typically over ad-hoc networks), where no guarantees regarding the availability and responsiveness of their constituting parts exist. Examples are systems of intelligent vehicle navigation, decentralized flight planning and healthcare monitoring. These systems feature a significant level of autonomy [24], which is connected with (self-) awareness [26] and (self-) adaptation [19] properties. At the same time, they feature a number of challenges in their development, as traditional software development approaches rely on static software architecture and static pre-defined behavior and fail to efficiently capture the dynamic architecture and support the overall development process.

In response to this problem, the new paradigm of *ensemble-based development* has been suggested to guide the development of large-scale adaptive systems operating in dynamic environments, termed Ensemble-Based Component Systems (EBCSs) [5]. These systems are typically composed of *autonomous service-components*, forming dynamic groups, which encapsulate knowledge, interaction, and goals specific to the groups. These dynamic groups of components are termed *ensembles* [10].

The goal of this paper is to point out the specific challenges related to the design of EBCSs and suggest ways to deal with them. First, EBCSs are described (section 2), then the challenges are articulated (section 3) and finally two "classic" and one novel approach are assessed on how well they deal with the identified challenges (sections 4 and 5). The last two sections present key related work (section 6) and conclude (section 7).

2. ENSEMBLE BASED COMPONENT SYSTEMS

Investigating ways to model and design systems based on the ensemble paradigm is the core of the European project ASCENS (Autonomic Service-Component ENSembles) [11]. A first attempt to formalize the concept of ensembles within ASCENS has led to the development of SCEL (Service-Component Ensemble Language) [8], a formal language for modeling component systems enabling them for further analysis and verification. Relying on the concepts of SCEL, DEECo (Dependable Emergent Ensembles of Components) component model [5, 13] has been conceived and is currently under development and refinement.

The goal of DEECo is to allow for building systems consisting of autonomous, self-aware, and adaptable components, which are implicitly organized in ensembles. To this end, DEECo suggests a slightly different way of perceiving a component than is common in component-based software engineering; i.e., as a self-aware unit of computation, relying solely on its local data that are subject to modification during the execution time. The whole communication process relies on data exchange among components (prescribed by ensembles), entirely externalized and automated within the

HotTopiCS'13, April 20–21, 2013, Prague, Czech Republic.

runtime. This way, the components have to be programmed as autonomous units, without relying on whether/how the distributed communication is performed, which makes them very robust and suitable for rapidly changing environments.

Apart from being perceived and articulated in terms of autonomous components and ensembles, EBCSs are built around three key ideas:

- The notion of *belief* and its explicit management have a central role. Every component in an EBCS operates upon its local "private" knowledge, which represents the component's view of the environment and of the other components. Since this knowledge is at any time subject to change by the runtime framework, which is responsible for mapping parts of one component's private knowledge body to knowledge bodies of other components, it is better viewed as the "belief" that a component preserves. In that sense, ensembles, being prescriptions of knowledge mappings between components, stand as the belief management mechanism.

- Component computation is performed *in isolation*. In EBCSs, there are no means for a component to explicitly communicate with others. Component communication is realized implicitly by knowledge exchange externalized from the components and performed by the underlying framework. Thus, every computation is necessarily performed within a component's boundaries, which strengthens the notion of component autonomy.

- Component bindings are *dynamic*. In EBCSs, there are no explicit bindings between components. Ensembles bind components implicitly by prescribing the appropriate knowledge exchange. However, ensembles are formed only when specific conditions hold in the system, not always. This dynamic nature of ensembles makes the architecture of EBCSs "emerge" during runtime.

As an example illustrating the main concepts of EBCSs, let us consider a system of intelligent vehicle navigation. The system consists of drivers, moving around a city in their "smart" vehicles. Drivers have to reach particular destinations within some time limits, which depend on their daily schedule. Vehicles are equipped with sensors of basic capabilities, e.g., monitoring the fuel and battery level of the car, but also more sophisticated ones, e.g., monitoring the traffic level along the route. Vehicles can only park and refuel in designated stations. They can also communicate with each

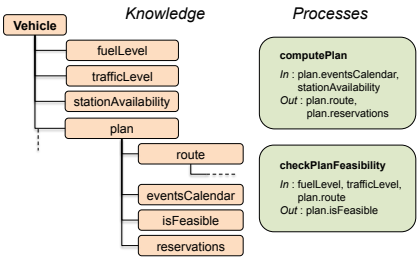


Figure 1: `Vehicle` component specification in DEECo.

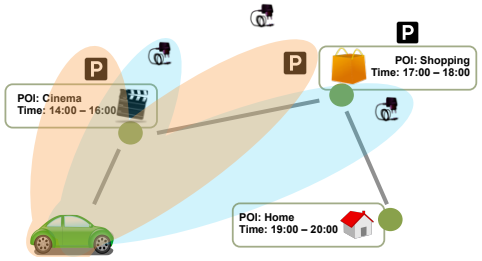


Figure 2: Possible ensembles between a `Vehicle` component and several `Parking/Refueling Station` components.

other and with the parking/refueling stations that lie within their transmission spectrum. No central coordination point is assumed; there is no global control or global planning. The whole system can be seen as a set of nodes which form dynamic communication networks (ad-hoc clouds) to serve a specific goal: vehicles should arrive at their destinations in time, leveraging the infrastructure resources in a close-to-optimal way.

When viewing the above system as an EBCS, the obvious candidates for (DEECo) components are vehicles, parking stations and refueling stations. For example, the `Vehicle` component can be specified in terms of its knowledge and processes as in Figure 1.

Possible (DEECo) ensembles are identified by looking at the different interaction scenarios among the components. For example, when a vehicle is close to a parking station, it may need to contact it to get informed about the available parking lots and reserve a lot if possible. This interaction is prescribed by one ensemble operating between a vehicle and (possibly multiple) parking stations, where the `Vehicle` obtains a belief over the lot availability information of the `Parking Stations`. A graphical representation of possible ensembles between a vehicle and parking or refueling stations is shown in Figure 2.

3. PROBLEM STATEMENT

It is clear that the number of different components and possible combinations of them in ensembles grows together with the magnitude of the system and the number and complexity of interactions we want to model. Consequently, even if we are able to *engineer* the above intelligent navigation system in terms of DEECo concepts, it remains challenging to *design* such a emergent system and retain some guarantees regarding its overall behavior and interactions.

This stems from the fact that DEECo concepts are rather low-level and focus primarily on supporting the implementation and deployment. They lack a broader system view that will take into account the system requirements and design alternatives. A broad, high-level view is crucial when dealing with systems of high complexity as it allows abstracting away from details of computation and interaction and reasoning about properties of the (distributed) system as a whole. Examples of interesting properties are performance-related ones, like communication overhead, information utilization, etc., and stability-related ones, like immunity to environmental changes, adaptability, robustness, etc.

Another issue is that it is problematic to map the architecture of the system (naturally comprising a number of components and ensembles) to the purpose it serves (its rationale). This is especially true for complex systems with numerous components and ensembles: in such cases tracing a low-level design decision, like the inclusion of a dynamic communication link in the system, back to its origin in the requirements analysis gets extremely difficult. At the same time, in such cases, specifying *why* an interaction has to take place is as important as specifying the interaction itself, as it allows for design justification and system predictability.

4. REQUIREMENTS MODELING

In order to be able to design and reason about an EBCS, we need to differentiate between stable and volatile information by obtaining a high-level view over the system. In this section, we will focus on approaches that capture the high-level behavior of a software system. For that, we need to draw our attention into the early phases of software development, such as the requirements analysis phase. It is thus necessary to examine prominent approaches in requirements modeling and assess their applicability in the domain of EBCSs.

A useful abstraction in Requirements Engineering (RE) is proven to be the *system goal*. A goal can be defined as a prescriptive statement of intent about some system whose satisfaction in general requires the cooperation of some agents forming the system. Goal-Oriented Requirements Engineering (GORE) [21] is concerned with the identification of system requirements through the elicitation and analysis of system goals.

Another useful abstraction is that of the *agent*. Agents are active components, i.e., with a choice of behavior, which may restrict their behavior to ensure the constraints that they are assigned. In GORE, agents are assigned responsibility for achieving goals.

In the rest of the section, the two most prominent approaches in GORE, KAOS and Tropos/i* methods are presented and discussed.

4.1 KAOS

KAOS is a goal-oriented requirements engineering methodology with a rich set of formal analysis techniques. KAOS stands for Keep All Objects Satisfied [15]. It is grounded on the following main ideas:

- The notion of *goals* has a prominent role during the requirements acquisition and analysis processes, offering the common advantages of goal-oriented approaches in RE [21].

- *Formal methods* are used *when* and *where needed* for RE-specific tasks. This allows different levels of expression and reasoning: semi-formal for modeling and structuring goals, qualitative for selection among alternatives, and formal for more accurate reasoning. This is possible, as each element modeled in KAOS has, in general, a two-level structure: the *outer, semantic* layer where the concept is declared together with its attributes and relationships to other concepts and the *inner, formal* layer for formally defining the concept. Formal reasoning can be used e.g., for checking goal refinement [7] and goal operationalization [16], conflict management [22] and obstacle (hazard,threat) analysis [23].

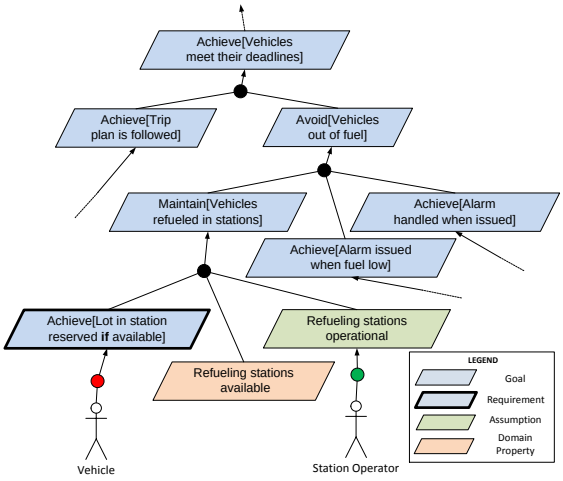


Figure 3: An excerpt of a KAOS goal model of the intelligent navigation case study.

A KAOS specification is a collection of complementary core models, which represent different views over the target system. In order to assess the applicability of KAOS models in the design of EBCSs, we will exemplify the process of deriving a KAOS specification on our intelligent navigation case study. The process spans four (practically interleaved) steps:

Goal elaboration step.

Goals are primarily obtained through the inspection of intentional keywords in natural language of stakeholders and by asking *why* and *how* questions about such statements. Goals are defined at different levels of abstraction: high-level goals capture global, strategic objectives; low-level goals capture local, technical objectives. After the elicitation of main goals, goals are organized into AND/OR refinement hierarchies with obvious semantics.

Goal refinement ends when every terminal goal is realizable by a single agent assigned to it. This means that the goal must be expressible in terms of conditions that are monitorable and controllable by the agent. In particular, a goal assigned to a software agent is a *software requirement*, whereas a goal assigned to an environment agent (e.g., a human agent) is an *expectation* or *assumption*.

Figure 3 depicts a possible goal refinement in the intelligent navigation case study, where the parent goal of having the vehicles refuel in the designated stations is decomposed into a requirement (vehicles reserve their places in the stations), an assumption (stations continue to operate) and a domain property (refueling stations are available). Domain properties are descriptive statements (as opposed to prescriptive ones, like goals or assumptions) that are explicitly captured in the goal model and serve for checking its completeness.

As an example of a complete goal specification, Figure 4 depicts the goal that vehicles should reserve their places in

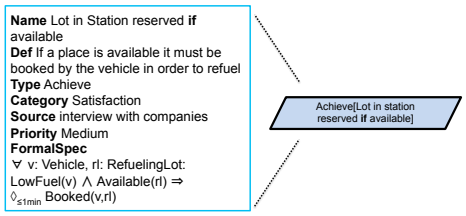


Figure 4: Formal specification of a goal in KAOS.

the refueling stations. Apart from its semantic layer, the goal's formal layer is captured in real-time Linear Temporal Logic (LTL). This enables the automatic verification of the goal model.

Agent modeling step.

An *agent* is an active object of the system, which acts as processor for operations. Agents can be software entities, but also environment entities, like human agents, devices, etc. They appear in the system in order to handle some requirement or assumption assigned to them during goal elaboration. For example, in Figure 3, the `Vehicle` and `Station Operator` agents are introduced. There is no single correct agent assignment; as with choosing among alternative goal refinements, assigning terminal goals to agents represents a design choice.

Object modeling step.

Objects are things of interest in the system whose instances may evolve from state to state. An object is modeled as an *entity*, *association* or *event*, depending on whether it is independent, dependent or instantaneous, respectively. Objects are derived by traversing the goal model and inspecting which entities are concerned in every goal. In our case study, examples of objects are the `Refueling station` entity, the `Reserved` association and the `Alarm issued` event.

Operationalization step.

The functions the agents need to employ in order to *operationalize* (fulfill) their assigned requirements is defined in terms of *operations*. An operation is an input-output relation over objects (specifically, object instances), whereas the application of the operation defines object creation and object state transition. Operations are derived both by goal fluents and from interaction scenarios, identified during requirements acquisition. Operations are declared by signatures over objects and have pre-, post-, and trigger conditions, specified in real-time LTL. As an example from our case study, a `Reserve lot` operation receiving a `Reservation request` object and creating the `Lot reservation` object could operationalize the "Lot in station reserved if available" requirement (Figure 3).

4.1.1 Discussion

Overall, KAOS is a well-developed methodology which exhibits the virtues of goal-oriented analysis in the field of requirements engineering. It successfully separates the stable (goals, assumptions, properties) from the volatile information (agents, operations) and provides a high-level view over the target system.

Another strong point of KAOS methodology is its formal reasoning support. When goals, domain properties, assumptions and operations are specified formally, the underlying models can be checked for consistency and completeness, typically by employing a SAT solver or theorem prover [17]. A semi-formal approach is also provided: during goal refinement and goal operationalization, the analyst can rely on instantiations of *formal refinement patterns* (extracted from a patterns catalogue) that are proven once and for all [7, 16].

+	1. captures the (intended) system behavior at a high level 2. allows for automatic formal reasoning
−	1. does not align requirements with architecture 2. is intended for requirements analysis and documentation, not system design

Table 1: KAOS positive and negative points.

A limitation of KAOS is that it does not provide a connection between requirements and architecture. Preliminary efforts towards this direction can be found in [14], where a process of deriving an architectural draft from goal, agent and operation models, is proposed. This draft is then iteratively refined based on instantiation of pre-defined architectural styles and patterns. This is a preliminary and rather generic attempt towards bridging the well-known gap between requirements and architecture [14]. When designing EBCSs, a tailored approach, dealing with the specific domain intricacies (high distribution, dynamism), seems more viable.

The main problem in directly applying KAOS towards meeting our goal of designing EBCSs is that the (classic) outcome of KAOS analysis is a Software Requirements Specification (SRS) document. While this serves the needs of requirements analysis, a successful design of complex, ensemble-based systems demands a mapping of the high-level models to lower-level ones and eventually to implementation artifacts.

The applicability of KAOS in the design of EBCSs is summarized in Table 1.

4.2 Tropos and i*

Tropos [3] is a methodology for building agent-oriented software systems that uses the i* modeling notation [25] (i-star refers to *distributed intentionality*). Tropos is based on the following key ideas:

- The notion of *agent* and related notions (e.g., goals, plans) are used in all software development phases, from early requirements down to actual implementation. To qualify as an agent, a software (or hardware) system is often required to have properties such as autonomy, social ability, reactivity, proactivity and rationality.
- The *early phases* of requirements analysis, i.e., the phases which precede the prescriptive requirements specification of the system-to-be, are considered equally important to the later phases and are thus elaborated in detail. This allows for a deeper understanding of the environment (organizational context) where the software must operate and facilitates the early resolution of conflicts between stakeholders.

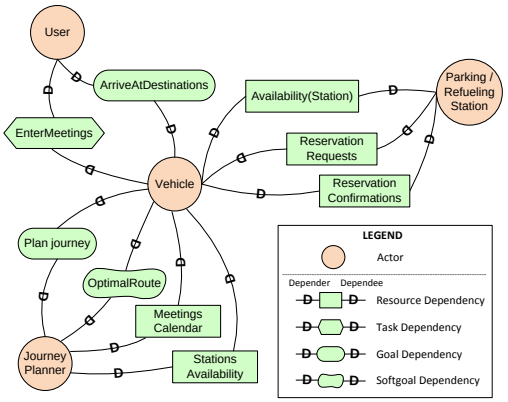


Figure 5: A possible i* Strategic Dependency model of the intelligent vehicle navigation case study.

The Tropos methodology spans four levels:

Early requirements.

The early requirements phase is concerned with the understanding of a problem by studying its organizational setting (the *system-as-is*). The output of this phase is an i* *Strategic Dependency* (SD) and an i* *Strategic Rationale* (SR) model. The SD model captures the relevant actors and their interdependencies in terms of goals to be achieved, tasks to be performed and resources to be furnished. The SR model determines through a means-end analysis how the goals can be fulfilled through contributions of other actors.

Late requirements.

During late requirements phase the *system-to-be* (target system) is described within its operational environment. The output of this phase is a requirements specification in the form of SD and SR models (refined versions of the early requirements phase models), which describes all functional and non-functional requirements of the system-to-be. The difference from the early requirements phase is that now the software to be developed comes into the picture as one or more intentional actors.

Figure 5 depicts a possible SD graph of our intelligent navigation case study at the late requirements phase. As depicted, the `User` actor depends on `Vehicle` to arrive at his destinations in time and, vice-versa, `Vehicle` depends on the fact that `User` will follow the routine of entering the meetings in the calendar. Dependencies exist between the `Vehicle` and the `Parking/Refueling Station` (modeled together for brevity) as well. `Journey Planner` has been introduced at this phase as part of the software to be developed. `Journey Planner` is delegated the responsibility of computing an (optimal) journey plan.

A means-end analysis of the `Vehicle` and the `Parking/Refueling Station` actors is depicted in the SR model of Figure 6. Specifically, the `Vehicle`'s goal of meeting the scheduled deadlines is decomposed into sub-goals, which are operationalized into tasks. As an example, the goal of being able to stop at the meeting points is satisfied by performing (in advance) reservations at the parking/refueling stations,

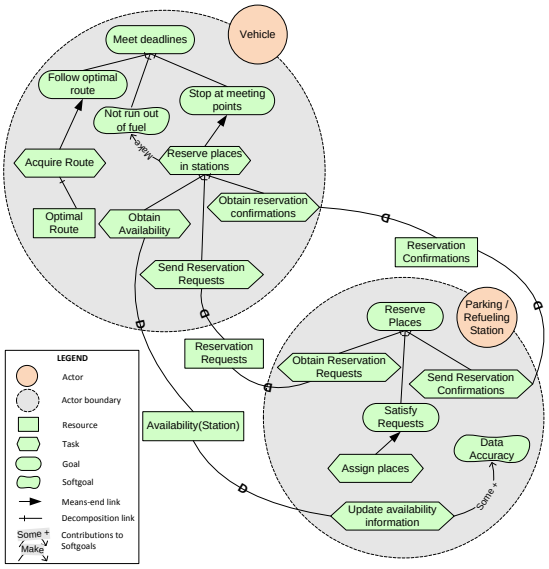


Figure 6: A possible i* Strategic Rationale model for `Vehicle` and `Parking/Refueling Station` actors.

which in turn is decomposed into the tasks of obtaining the station availability information, requesting a place in the relevant stations, and receiving reservation confirmations. At the `Station`'s side, the identified goals and tasks are decomposed in a similar manner.

Architectural design.

Architectural design defines the system's global architecture in terms of sub-systems, interconnected through data and control flows. The SD and SR models derived from previous levels are further refined. The refinement – inclusion and removal of actors and dependencies – is determined by a quality analysis (analysis and refinement of softgoals). The quality analysis guides the selection between alternative architectural styles from a catalogue of organizational styles (pre-defined in terms of the i* concepts of actors, dependencies, goals and tasks) [9]. For example, if during the requirements phase of our intelligent navigation case study the "response time" softgoal was identified, we could choose an organizational style which ensures low response time, e.g., the "joint venture" or the "pyramid" style.

After the refinement of the actor and goal models, the *capabilities* needed by the actors to fulfill their goals and plans are identified by inspecting the extended SD (actor) diagram, presuming that each actor's dependency relationship can give place to one or more capabilities triggered by external events.

As a final step, a set of *system agents* is identified and for each of them one or more capabilities are assigned. In general, the process of assigning capabilities to agents is not unique and depends heavily on the designer's view of the system (in terms of agents). Nonetheless, Tropos offers a set of pre-defined social patterns recurrent in multi-agent literature like `Bidding`, `Broker`, `Matchmaker`, `Embassy`, etc., to guide this process [9].

Detailed design.

Detailed design deals with the specification of the agents' micro level, that is, the agents' behavior and communication. During detailed design, multiple *capability* and *plan diagrams* are created. Additionally, Agent-UML (AUML) sequence diagrams are employed to specify the interaction protocols.

Following the detailed design, a *concrete implementation* of the system is produced by using one of the agent-oriented development environments. For example, JACK Intelligent Agents [20] can be employed, which stands as a reification of the conceptual Belief-Desire-Intention (BDI) [18] agent model in Java programming language.

4.2.1 Discussion

The main contribution of Tropos methodology is that it tries to align requirements analysis with system design and implementation. Its novel idea is to base such an alignment on early requirements concepts, such as actors, goals and plans, rather than implementation-level concepts, like classes and methods. A strong point of Tropos is that it provides a small and manageable set of knowledge level notions, which are used throughout the software development process.

At the same time, the biggest shortcoming of this methodology is the plethora of design phases and models (actor/goal models, capability/plan/sequence models, BDI agent model), which complicates the design process. Moreover, the transition between the phases is in most cases manual and relies heavily on subjective design decisions. In the design of complex EBCSs the automatic transition between abstraction levels is a necessity and cannot be overlooked.

To conclude, the direct application of Tropos method in the design of EBCSs would eventually result into mapping one or more system agents to (DEECo) components and agent communication to (DEECo) ensembles. However, current agent development frameworks [2, 20] assume static bindings between the system actors and therefore fail to cope with the dynamic, emergent architecture (captured in terms of ensembles), which is one of the key characteristics of EBCSs.

The applicability of Tropos in the design of EBCSs is summarized in Table 2.

+	1.	aligns the requirements phase with architecture and implementation phases
	2.	preserves a manageable set of concepts throughout the software development phases
−	1.	comprises a number of models with manual mappings between them
	2.	does not cope with the emergent architecture of EBCSs

Table 2: Tropos positive and negative points.

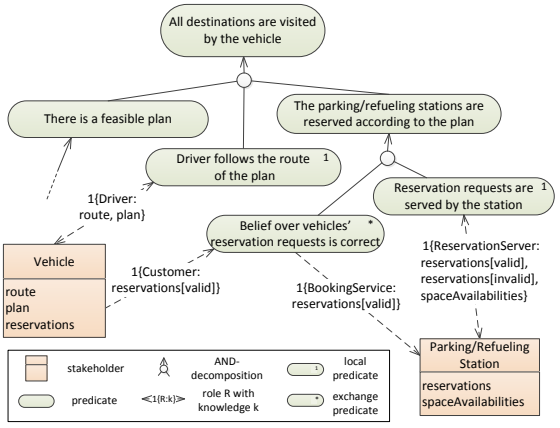


Figure 7: A partial decomposition of the main system predicate in the intelligent mobility case study.

5. TOWARDS AN ENSEMBLE-BASED DESIGN METHOD

Having evaluated KAOS and Tropos methods, it is clear that they are not directly applicable in the design of EBCSs. On their background, in this section we will describe a novel design method termed Predicate Refinement Method (PRM) [4], which employs similar concepts. We will also justify why PRM successfully deals with the intricacies of EBCSs, by accounting for what is missing in KAOS and Tropos.

PRM builds upon the idea of iterative refinement of system specification, employed in goal-oriented requirements engineering. The main goal of PRM is to complement DEECo low-level concepts with design-level abstractions that will allow for a) design-time reasoning, and b) automatic preparation of DEECo artifacts.

PRM is based on capturing the high-level system goals expressed in terms of propositional claims (predicates). It consists of three levels (phases): *system level* design, *ensemble level* design, and *component level* design, followed directly by implementation.

As a starting point, at the system level, PRM elaborates on the questions "which (global) goals must be achieved?" and "which system attributes must be maintained?". The next question is "who is responsible for achieving/maintaining these attributes?". Answering these questions yields the system's initial *stakeholders* and interaction *predicates*.

A stakeholder is a participant/actor of the system. Each stakeholder is defined in particular by its knowledge (i.e., its domain-specific data). For example, when applying the method in the running case study (Figure 7), the `Vehicle` and `Parking/Refueling Stations` stakeholders are identified. At this level, the system goals are expressed by predicates over the stakeholders and their knowledge; each involvement of a stakeholder in an predicate is a *role* of the stakeholder in the predicate. Unlike i*, PRM does not capture the dependencies of actors at a strategic level, but at the lower level of data that has to be exchanged and "believed"; stakeholders contribute their knowledge (interfaced by roles) for the assessment of each predicate they take a role in. Predicates represent system properties that should hold over the whole system lifetime. In an idealized system, predicates would become system *invariants*, whereas in a real system predicates should hold "frequently enough". As opposed to goals in KAOS, predicates have a more descriptive nature, and typically refer to the present state, not some future actions.

After identifying all top-level predicates, the process con-

tinues into refining them into sets of sub-predicates. The refinement is essentially an AND-decomposition, with the conjunction of sub-predicates implying the parent predicate. The iterative refinement process in PRM ends once all the leaf predicates are directly mappable to DEECo computational or communication semantics, that is, to component and ensemble processes. In particular, a predicate needs no further decomposition when a) it involves a single stakeholder and can be ensured by manipulation of this stakeholder's knowledge (via an underlying component process) – *local predicate* – or b) the predicate involves exactly two stakeholders and can be ensured by mapping one stakeholder's knowledge part(s) to the other (via an underlying knowledge exchange mechanism) – *exchange predicate*.

Figure 7 depicts a partial decomposition tree of the top-level predicate of having all destinations visited by the `Vehicle`. This predicate is refined into three "necessities", independent to each other: the necessity a) to have a feasible plan, b) to follow it, and c) to have places in stations reserved accordingly. Following the plan's route is essentially a local predicate, as it involves the `Vehicle` stakeholder only. In contrast, the predicate of having a correct belief over the reservation requests (Figure 7) is an example of an exchange predicate, as it involves both the `Vehicle` and the `Station` stakeholder.

At the next phases of PRM, the predicates are transformed to precise system specifications. Specifically, during *ensemble level design*, the exchange predicates are turned into DEECo ensembles, by specifying the necessary condition for knowledge exchange along with the information of which knowledge parts are going to be exchanged. In *component level design*, finally, stakeholders are turned into DEECo components, comprising the stakeholders' knowledge and "operationalizing" (in terms of KAOS) the local predicates they are involved in by means of (DEECo) processes.

Compared to PRM, KAOS and Tropos address well the high-level modeling, however naturally they do not provide constructs for alignment with the emergent architecture of EBCSs. This is summarized in Table 3. Further, to complete this picture, we evaluate below in more detail how PRM addresses the three fundamental characteristics of EBCSs.

Belief handling.

The semantics of the *belief* that components in EBCSs preserve about the other components and the environment is what remains stable throughout the system phases (over the whole lifetime of the system) and provides an effective way to reason about global properties of interest. In PRM, we explicitly capture this semantics at a higher level by modeling the knowledge flow in terms of knowledge that has to be distributed and "believed". Moreover, the mapping of exchange predicates to ensembles provides a convenient belief managing mechanism. Although the concept of belief is not new (agent-based approaches like Tropos employ the same concept in the design of agent-based architectures, like BDI), the novelty of PRM lies in featuring belief as the base abstraction that aligns requirements with architecture and implementation phases.

Isolated computation.

In order for component autonomy to be reified, components in DEECo perform their tasks in isolation. This is reflected in the design by introducing local predicates, which are mapped to component processes as the process progresses. In that sense, PRM captures both the requirements of the whole system (higher-level predicates) and of isolated components (local predicates), in contrast with requirements elicitation a là KAOS, which is necessarily system-wise.

	KAOS	Tropos	PRM
high-level modeling	+	+	+
req/ments & architecture alignment	–	+	+
dynamic, emergent architecture support	–	–	+

Table 3: A comparison of PRM with KAOS and Tropos methods.

Dynamic component links.

Dynamic component links, reified by ensembles in DEECo, provide the way to deal with the emergent architecture of EBCSs. In PRM, dynamic links are reflected by exchange predicates. This property of EBCSs is not directly supported either by KAOS or Tropos methods.

6. RELATED WORK

As we are not aware of any work that combines the ensemble paradigm with goal-oriented requirements analysis to devise a design method for EBCSs, in this section we will refer to general approaches towards the requirements modeling and design of EBCSs. We have already extensively described, in section 4, two prominent approaches in goal-oriented modeling and design, namely KAOS and Tropos/i* methods.

Recent work in requirements modeling and in particular targeting the domain of EBCSs has been carried out within the scope of the ASCENS project and has been integrated into Statement of the Affairs (SOTA) [1], General Ensembles Model (GEM)[12] and POEM [11] models.

SOTA is concerned with the overall domain and the requirements of the system. The key idea is to abstract the behavior of a system with a single trajectory through a *state space* (or state-of-the-affairs space), which represents the set of all possible states of the system at a single point of time. Similarly to PRM, 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 a goal pre-condition, its post-condition, and utilities (non-functional requirements or constraints). To support adaptation, SOTA relies on the goal model to help understand according to which (self-adaptive) architectural scheme a system should be architected so goals can be achieved [6].

For a more detailed specification of behaviors and goals, POEM [11] model has been proposed and is currently at a rather preliminary stage. Finally, GEM [12] stands as a formal system model which represents ensembles as relations that describe the complete behavior of the ensemble over its lifetime. GEM is intended to serve as a semantic foundation for various kinds of calculi and formal methods that often have a particular associated logic.

7. CONCLUSION

In this paper, the specific characteristics of systems based on the ensemble paradigm have been described. The challenges related to the design of EBCSs have also been identified. A successful design method should be able to capture the high-level objectives of the system under consideration in a model amenable to design-time analysis, and, at the same time, simplify the transition towards architecture-level models, like DEECo model. As possible ways to construct models of the system at a high-level we have looked into two prominent methods in requirements modeling, KAOS and Tropos/i*. They both possess their strong points but also feature serious limitations, when employed in the design of EBCSs. In response to that, a novel method based on the iterative refinement of system requirements expressed by predicates on stakeholders' knowledge was outlined. The novelty of the proposed method lies in reasoning along the line of what needs to hold in the system at every time instant (predicates), instead of what needs to be performed (actions) or achieved (goals).

As future work, we plan to work on a prototype implementation of PRM, which will exhibit the advantage of deriving the system components and ensembles from requirements in a (semi-) automatic way. We also plan to formalize the concepts of predicate and predicate refinement in such way that will allow for formal verification of the design model.

8. ACKNOWLEDGMENTS

This work is a part of RELATE project supported by the European Commission under the Seventh Framework Programme FP7 with Grant agreement no.: 264840ITN.

9. REFERENCES

[1] D. Abeywickrama et al. SOTA: Towards a General Model for Self-Adaptive Systems. In *WETICE '12*, pages 48–53. IEEE CS, 2012.

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

[3] 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.

[4] T. Bures et al. Language extensions for implementation-level conformance checking. ASCENS Deliverable D1.5, 2012.

[5] T. Bures et al. DEECo – an Ensemble-Based Component System. Tech. Rep. D3S-TR-2013-02, D3S, Charles University in Prague, 2013.

[6] G. Cabri, M. Puviani, and F. Zambonelli. Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles. In *CTS '11*, pages 508 –515, May 2011.

[7] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. *SIGSOFT Softw. Eng. Notes*, 21(6):179–190, Oct. 1996.

[8] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese. A Language-based Approach to Autonomic Computing. In *FMCO '11*, volume 7542 of *LNCS*, pages 25–48. Springer-Verlag, 2012.

[9] P. Giorgini et al. A Goal-Based Organizational Perspective on Multi-Agents Architectures. In *ATAL '01*, Seattle, USA, Aug. 2001.

[10] M. Holzl et al. Engineering of Software-Intensive Systems: State of the Art and Research Challenges. In *Software-Intensive Systems and New Computing Paradigms*, volume 5380 of *LNCS*, pages 1–44. Springer Berlin / Heidelberg, 2008.

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

[12] M. Holzl and M. Wirsing. Towards a System Model for Ensembles. In *Festschrift in honor of Carolyn Talcott*, volume 7000 of *LNCS*. Springer, 2011.

[13] J. Keznikl, T. Bures, F. Plasil, and M. Kit. Towards Dependable Emergent Ensembles of Components: The DEECo Component Model. In *WICSA/ECSA '12*. IEEE CS, Aug. 2012.

[14] A. V. Lamsweerde. From System Goals to Software Architecture. In *FSM '03*, volume 2804 of *LNCS*, pages 25–43. Springer-Verlag, 2003.

[15] A. V. Lamsweerde and E. Letier. From Object Orientation to Goal Orientation: A Paradigm Shift for Requirements Engineering. In *RISSEF '12*, volume 2941 of *LNCS*, pages 325–340. Springer-Verlag, 2004.

[16] E. Letier and A. van Lamsweerde. Deriving operational software specifications from system goals. *SIGSOFT Softw. Eng. Notes*, 27(6):119–128, 2002.

[17] C. Ponsard, P. Massonet, J. F. Molderez, A. Rifaut, A. V. Lamsweerde, and H. T. Van. Early verification and validation of mission critical systems. *Form. Methods Syst. Des.*, 30(3):233–247, June 2007.

[18] A. S. Rao and M. P. Georgeff. BDI Agents: From Theory to Practice. In *ICMAS '95*, pages 312–319, 1995.

[19] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[20] A. O. Software. JACK Intelligent Agents Manual, Release 5.3. http://www.agent-software.com, 2005.

[21] A. van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *RE '01*. IEEE CS, 2001.

[22] A. van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Trans. Softw. Eng.*, 24(11):908 –926, Nov. 1998.

[23] A. van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Trans. Softw. Eng.*, 26(10):978–1005, Oct. 2000.

[24] E. Vassev and M. Hinchey. The Challenge of Developing Autonomic Systems. *Computer*, 43(12):93 –96, Dec. 2010.

[25] E. Yu. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In *RE '97*, pages 226–. IEEE CS, Jan. 1997.

[26] F. Zambonelli et al. On Self-Adaptation, Self-Expression, and Self-Awareness in Autonomic Service Component Ensembles. In *SASOW '11*, pages 108 –113, Oct. 2011.

4.3 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 ACM 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

This paper, published as [KBP+13], provides one of the main contributions of the thesis, which is an architecture design method for siCPS. The focus of this paper is on the domain of *dynamically evolving resilient distributed systems* (RDS), which was casted into the domain of software-intensive Cyber-Physical Systems (siCPS) in subsequent works. As in the paper included in Section 4.2, the term *ensemble-based component systems* (EBCS) is used to refer to siCPS modeled according to the ensemble paradigm. EBCS is used throughout the paper as the reference architecture model of the proposed design method, whereas DEECo (Section 2.2.2.3) is used as a concrete instance of EBCS.

The main idea of the paper is to tackle the challenge of systematic architecture design of RDS that comply to the EBCS reference model. Due to the dynamic and stateless nature of the ensemble concept, it is problematic to determine a proper EBCS architecture (comprising components, component) from the system-level goals and requirements. This stems from the conceptual gap between the high-level system goals and the relatively low-level computation and communication constructs as featured by EBCS.

In response to this challenge, a novel design method, the *Invariant Refinement Method* (IRM), is introduced and elaborated. The goal of IRM is to guide the refinement of high-level goals to low-level architecture concepts so that the compliance of design decisions with the overall system goals is captured, validated, and – if possible – formally verified. As such, IRM directly addresses the objective **O2**.

IRM builds on the idea that siCPS have to maintain a certain level of *operational normalcy*, i.e. the property of being within certain bounds that define a range of normal operation. As a result, a component process or an ensemble in an EBCS architecture can be expressed in terms of the particular operational normalcy it maintains. IRM features the concept of *invariant* to express this normalcy in terms of knowledge evolution of the associated components or group of components. At the same time, invariants in IRM are also used to describe high-level system goals. The objective of the design process becomes then to link the high-level invariants to low-level ones via a series of decomposition steps with formal refinement semantics. At each refinement, the conjunction of sub-invariants implies the parent invariant.

Since a formal refinement of invariants is a relatively complex task, the paper also proposes a formal framework for invariant refinement, which is the second main contribution. The framework relies on defining *invariant patterns* capturing the specifics of invariants at different abstraction levels and using them in the definition of rules that govern the decomposition at the same level or across adjacent levels of abstraction.

Comments on Authorship

Overall, the main idea of the paper is of equal authorship. I personally contributed to the elaboration of the idea and the technical details, and in the positioning of the work in the context of ensemble-based component systems and against the related work. I was also responsible for the design of the case study of cooperative e-vehicles used in the evaluation of this work.

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.

CBSE'13, June 17–21, 2013, Vancouver, BC, Canada.

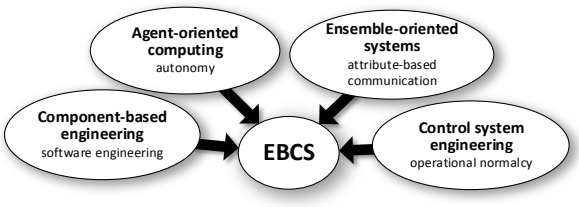


Figure 1: Context of Ensemble-Based Component Systems (EBCS).

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.

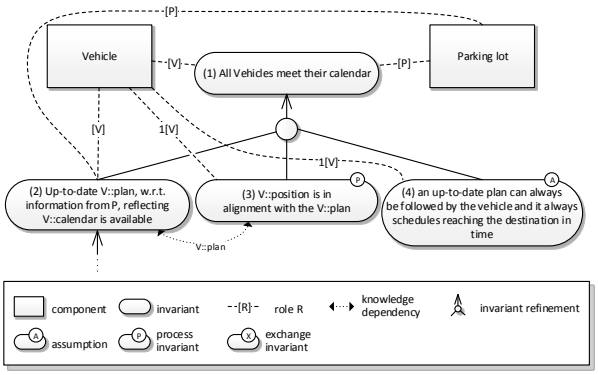


Figure 3: Top-level design of the case study.

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

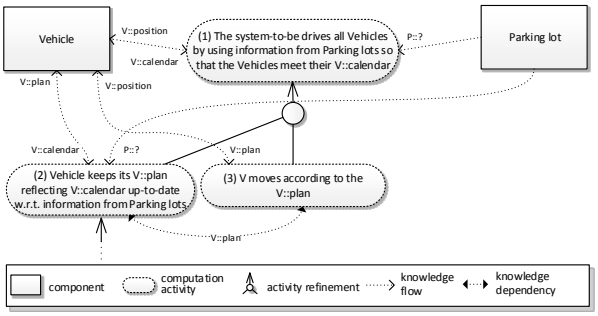


Figure 4: Dual, computation-activity-based view on the top-level design of the case study from Figure 3.

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. $I_{s1} \wedge \dots \wedge I_{sn} \not\Rightarrow false$ (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 $\text{V::availabilityList} = B_{\text{Vehicle}}^{\text{Parking lot}}(\text{P::availability})$).

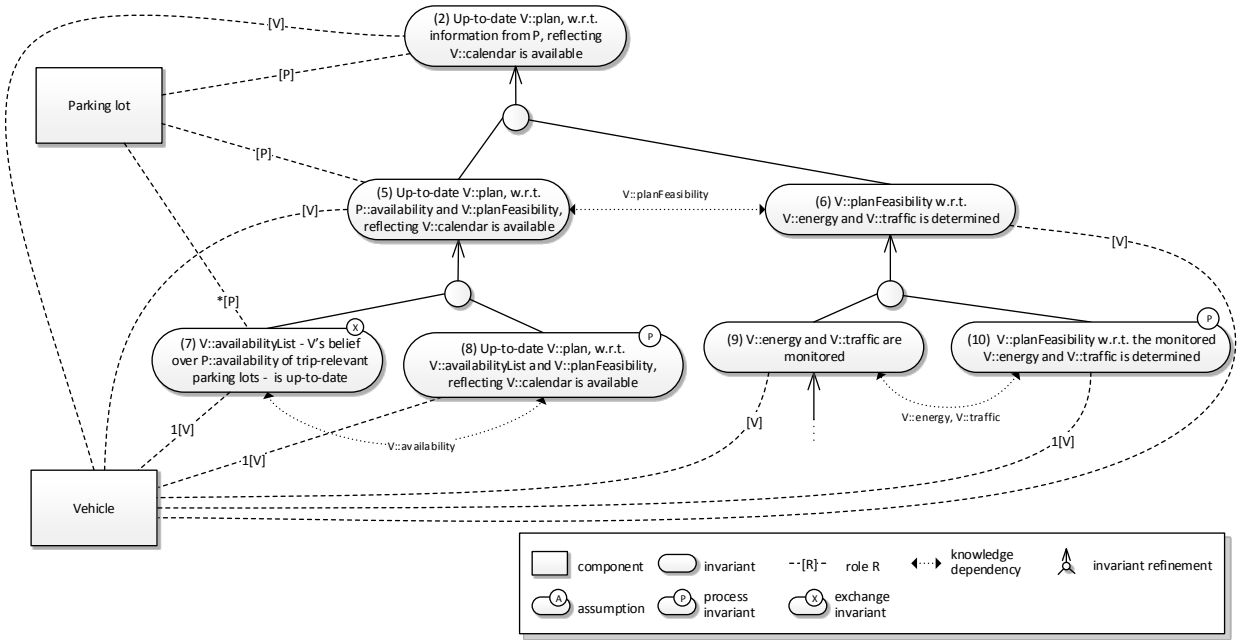


Figure 5: Invariant refinement of "V has an up-to-date V::plan reflecting V::calendar".

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*

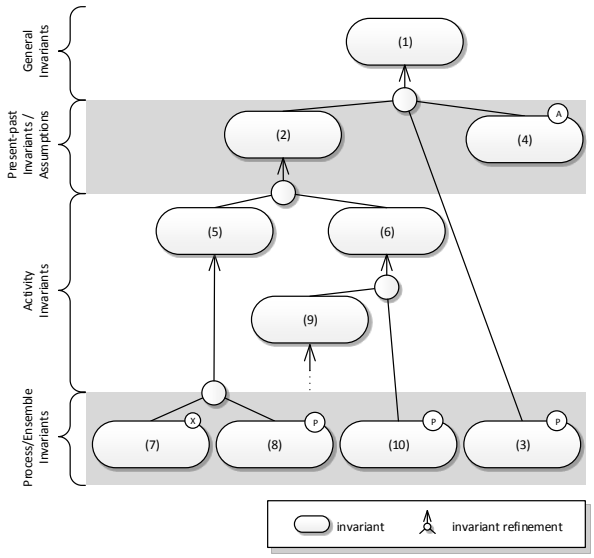


Figure 6: Patterns of invariants in the case study.

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.

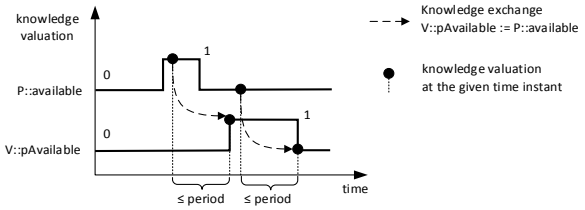


Figure 7: Example of knowledge evolution in time when employing (periodic) knowledge exchange.

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):

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

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:

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

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:

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

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:

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

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:

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

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:

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

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

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

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 $I_1, \dots, I_n$ (which means that by definition $I_1, \dots I_n \Rightarrow I$), we can distinguish 4 different cases:

(1) All $I_1, \dots, I_n$ hold and I holds: Correct operation of the system.
(2) All $I_1, \dots, I_n$ hold and I does not hold: Error in design – mostly because of neglecting a hidden assumption in refinement of I.
(3) At least one $I_1, \dots, I_n$ 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 $I_1, \dots, I_n$ 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.

4.4 Formalization of Invariant Patterns for the Invariant Refinement Method

Tomáš Bureš,
Ilias Gerostathopoulos
Jaroslav Keznikl,
František Plášil,
Petr Tůma

In the **Software, Services and Systems**.

Volume 8950 of Lecture Notes in Computer Science,
published by Springer International Publishing,
pages 602-618,
print ISBN 978-3-319-15544-9,
online ISBN 978-3-319-15545-6,
2015.

The original version is available electronically from the publisher's site at http://dx.doi.org/10.1007/978-3-319-15545-6_34.

Summary of the Paper

This paper, published as [BGK+15], is a direct continuation of the research work on the *Invariant Refinement Method* (IRM), introduced in the paper included in Section 4.3; it also contributes to meeting the objective **O2**. As in that paper, the focus is on the domain *ensemble-based systems*, i.e. software-intensive Cyber-Physical Systems modeled according to the ensemble paradigm (Section 2.2.2).

The main idea of the paper is to elaborate on the formal framework of IRM (proposed in the paper of Section 4.3) that guides the decomposition of higher-level invariants into conjunctions of lower-level sub-invariants. At each decomposition, the conjunction of sub-invariants implies the parent invariant. This complies with the traditional interpretation of *refinement* in software engineering, where the products of refinement (sub-invariants, sub-components, etc.) collectively exhibit the behavior of the refined element (parent invariant, composite component, etc.) and potentially more. In order to help the designer in the invariant refinement process, a set of *patterns* of invariants at different levels of abstraction, and a number of rules to transition between them have been previously proposed (see Section 4.3). In this paper, the invariant patterns are formalized, and formal means are given to decide upon the correctness of the refinement. Following the formally proven refinement rules automatically endorses an IRM design with "correctness by construction" guarantees, and adds to the overall dependability of the underlying ensemble-based system.

Technically, each of the five identified invariant patterns – general, present-past, activity, process, and exchange invariant – are defined as logic predicates (in a second order predicate logic with arithmetic) according to the knowledge evolution that they describe. Correctness by construction is provided by establishing the relations that have to hold between invariant patterns involved in a correct decomposition in the form of mathematical proofs in the aforementioned logic. An important proof is that of a recurrent type of decomposition termed *pipeline decomposition*. In a pipeline decomposition the children reflect simple pipeline-like flows among the corresponding activities that refine the parent activity. Finally, more complex refinement types are discussed and common mistakes that arise in their formal proofs are identified.

Comments on Authorship

In this paper, I personally contributed in the elaboration and refinement of the definition of the different invariant patterns and their formalization, and of the formal proofs for their correct decomposition. I was also responsible for positioning the work against the related work, in particular that of KAOS, Tropos, and UML 2 interaction diagrams.

Formalization of Invariant Patterns for the Invariant Refinement Method

Tomáš Bureš, Ilias Gerostathopoulos, Jaroslav Keznikl,
František Plášil, and Petr Tůma

Charles University in Prague
Faculty of Mathematics and Physics
Prague, Czech Republic
{bures,iliasg,keznikl,plasil,tuma}@d3s.mff.cuni.cz

Abstract. Refining high-level system invariants into lower-level software obligations has been successfully employed in the design of ensemble-based systems. In order to obtain guarantees of design correctness, it is necessary to formalize the invariants in a form amenable to mathematical analysis. This paper provides such a formalization and demonstrates it in the context of the Invariant Refinement Method. The formalization is used to formally define invariant patterns at different levels of abstraction and with respect to different (soft) real-time constraints, and to provide proofs of theorems related to refinement among these patterns.

Keywords: architecture refinement, requirements, assume-guarantee

1 Introduction

Invariant-based design is advantageous for designing adaptive self-organizing systems formed by ensembles of autonomic components [7–9] – see e.g. SOTA [1] – as it explicitly captures the valid states of the system, i.e., the invariant properties of a correct system. Such ensemble-based systems [2] operate autonomously in an open-ended environment, and invariants are well-suited for capturing the properties of a component with respect to its environment.

The problem of invariant refinement is that the requirements of a system are typically described in a much higher level of abstraction than the properties (invariants) of the individual constituents of system architecture (components, component processes, ensembles). The transition from high-level obligations to low-level constraints includes a number of design choices without firm borders and guidelines, and thus is prone to errors.

In our work we have proposed to bridge this gap by gradual step-wise refinement (decomposition) of invariants, which ends up with detailed specification of the behavior of the involved architectural elements – ensembles, components. We call this approach *Invariant Refinement Method – IRM* [2, 10]. IRM however requires the steps of the refinement to be well-defined (ideally formally), so that the refinement itself represents a proof of the correctness of the design. *In*

other words, it is necessary to have (formal) means allowing for deciding upon the correctness of the refinement.

Having a formal framework that formalizes these relations allows for (i) design-time guarantees of design correctness, i.e., guarantees that the system design truly addresses the high-level requirements, and (ii) runtime monitoring, i.e., detection of discrepancies in system design during execution.

In this paper we provide such a formal framework, and also provide mathematical proofs of "correctness by construction", as a continuation of the work presented in [10]. To do so, we first describe and formalize the invariant concept and invariant refinement in the light of our running example (Section 2). We then provide a formal account of the invariant patterns that can guide the IRM design (Section 3), and provide the main contribution of the paper, i.e., the set of theorems and lemmas that formally ground the relations between the invariant patterns (Section 4). Finally, we discuss some of the implications of our approach and conclude (Section 5).

Personal Note: Ideas presented in this paper have been inspired by the work of Martin Wirsing in the field of formal software engineering of autonomous service-components. We have known Martin for a long time, and we have been able to stay up-to-date with the advancements of his research group at LMU, as one of the authors has been a visiting professor at LMU for the past years. We have also had the opportunity to work with him and his colleagues from his department in the ASCENS project, which he was coordinating. Cooperating with Martin is always both enjoyable and inspiring, not only because of his firm knowledge and fresh ideas, but also because of his kind and welcoming personality.

1.1 Running Example

To illustrate the IRM-based design, we use a running example from the ASCENS e-mobility case study [14]. In this case study, electric vehicles (e-vehicles) have to coordinate in order to reach particular places of interest (POIs) within certain time constraints specifying the expected POI arrival and departure times, as prescribed by the drivers' daily schedules (calendars). At the same time, e-vehicles compete for stopovers in limited energy charging stations (CSs) along their route. Specifically, each e-vehicle has to plan its individual trip according to the driver's calendar and the (perceived) available time slots for charging at each relevant charging station. This results in a fully decentralized – and thus scalable – system.

To simplify the presentation of our approach, we assume for the running example that each vehicle has a single driver and a single destination POI. This results in the scenario where the goal of every vehicle is to reach its POI in time, while visiting charging stations during the trip if necessary. The charging stations may however become unavailable at any time and thus it is necessary to introduce monitoring of charging stations and potential re-planning.

2 Background

2.1 Invariant refinement

In principle, IRM employs *invariants* to describe a desired state of the system-to-be at every time instant; i.e., to describe the *operational normalcy* of the system-to-be, essential for its continuous operation. When using IRM to design ensemble-based systems, the objective is to refine the overall system goal(s) in an iterative way and end up with the invariants that concern the individual constituents of system architecture – components, component processes, and ensembles.

The refinement is performed by decomposing a higher-level invariant into a set of lower-level sub-invariants (AND-decomposition). In order for the decomposition of a parent I_p into the children $I_{s1}, \ldots, I_{sn}$ to be an actual refinement, the *conjunction* of the children have to *entail* the parent, i.e., it has to hold:

$$I_{s1} \wedge \ldots \wedge I_{sn} \Rightarrow I_p \quad (entailment)$$
$$I_{s1} \wedge \ldots \wedge I_{sn} \not\Rightarrow false \quad (consistency)$$

This type of decomposition is applied iteratively, starting from the high-level invariants that reflect system-level goals and ending with low-level ones that refer to a single component or an ensemble of components. The outcome is a graph capturing the structural elaborations and design decisions at different abstraction levels. Since each decomposition step may involve a design decision, it is important to ensure that this decision complies with the entailment and consistency conditions.

Invariant refinement of the running example. An invariant-based design of a system targeting the running example is presented in Figure 1. A description of each individual invariant follows.

(1) This is the main goal of the scenario.
(2) This expresses a specific requirement on the designed system and the vehicle's planner input in particular. In this context, a plan is a black-box giving for each time instance the expected position of the vehicle at that time.
(3) This reflects the assumption that the plan is always realistic (i.e., that it is actually possible to follow it given the traffic and car characteristics), and that the driver would follow it precisely.
(4) This expresses the assumption that charging station availability does not change too quickly and that the initial set-up of the environment is "planning-friendly".
(5) A specific system requirement that constrains the input and timing of the planner. In particular, we assume read consistency with respect to the belief (i.e., new plan is always based on the *same or newer* belief than the previous plan). Moreover, (5) and (6) together represent the design decision of dividing the activity of computing the plan from remote data into two activities of (i) creating a local belief of the remote data and (ii) computing the plan from the local belief.

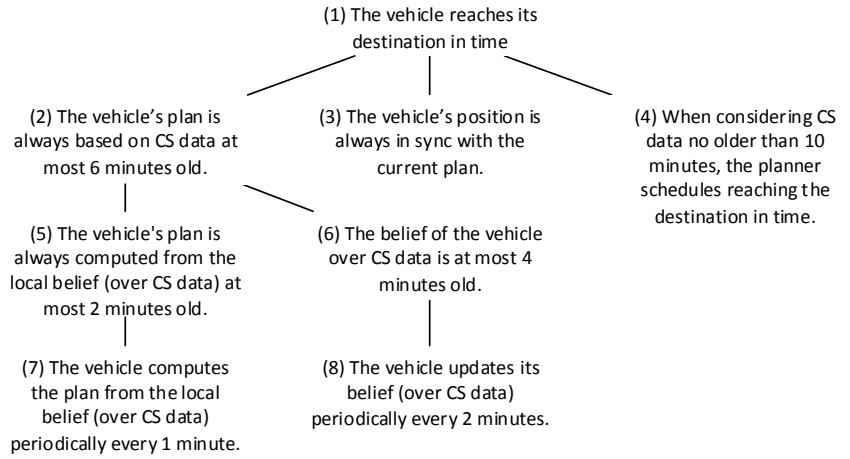


Fig. 1. Invariant refinement of the running example.

(6) A specific system requirement that constrains the timing of charging station monitoring and belief updating.
(7) A specific system requirement precisely determining the input and timing of the planner. In particular, we assume real-time periodic computation.
(8) A specific system requirement precisely determining the timing of CS monitoring. In particular, we assume (distributed) real-time periodic monitoring.

Note that the invariant-based design such as the one presented in Figure 1 is hardly ever a product of a top-down design process. In practice, a mixed top-down/bottom-up process is followed, where sub-invariants are identified by asking "*how* can this invariant be satisfied" and parent invariants are identified by asking "*why* should this invariant(s) be satisfied".

2.2 Invariant formalization

In general, the goal of invariant-based system design is to formally capture properties of a valid system. Thus, we will first discuss the necessary characteristics of such formalization (i.e., characteristics implied by the domain).

In the domain of (soft) real-time component ensembles, the way of expressing properties of a valid system is, as indicated by the running example, to capture a valid evolution of knowledge values in time. To do that, the underlying formalism has to provide means for referring to knowledge values at arbitrary time instants. When generalized, we can say the formalism needs to refer to timed sequences of knowledge values (i.e., timed streams of data), which provide a complete view on the knowledge value evolution in time.

This is explicitly formalized in the following definitions, where we consider time to be a non-negative real number, i.e., $\mathbb{T} \stackrel{\text{def}}{=} \mathbb{R}_0^+$.

Definition 1. *(Knowledge and its valuation)* Knowledge *is a set* $K = \{k_1, \ldots k_n\}$ *of knowledge elements, where the domain of* k_i *is denoted as* V_i. Knowledge valuation *of element* k_i *is a function* $\mathbb{T} \to V_i$ *which for each time* t *yields a value of* k_i *(denoted* $k_i[t]$*).*

Definition 2. *(Invariant) An* invariant *is a predicate (in a higher-order predicate logic with arithmetic) over knowledge valuation and time.*

In general, an invariant may refer to the knowledge valuation at an arbitrary time point/interval.

As further illustrated by the running example, when formalizing system design, it is critical to introduce formal assumptions about the environment of the system. Although this is often omitted in informal design approaches, without explicit assumptions the formalized system design is neither complete nor correct. Thus we differentiate between two types of invariants:

- *System invariants* reflect properties of the individual architectural elements of the system. Their validity is to be ensured by the implementation of the system.
- *Assumptions* reflect the properties of the system's environment assumed by system invariants. Validity of these invariants is usually out of control of the designer and is necessary for correct operation of the implementation.

For example, invariant (2) from the running example is a system invariant while invariant (4) is an assumption.

3 Invariant patterns

In general, the form of invariants is not explicitly restricted. However, at particular levels of abstraction (when describing architectural elements) there are several patterns virtually omnipresent in any invariant-based design [10]. It is thus beneficial to have means for concise and consistent representation of such invariant patterns.

General invariants. At the highest abstraction level, *general invariants* relate to system-level goals. They capture the operational normalcy of a system by relating the past and current knowledge valuations to future knowledge valuations. Therefore, a general invariant can have an arbitrary internal structure.

Present-past invariants. At a lower abstraction level, the invariants express that some knowledge is based on other knowledge, which, at the same time, is no older than a particular time interval – *lag*. This reflects the fact (abstracted by general invariants) that software systems cannot employ future knowledge to

maintain their operational normalcy, but have to depend on present and/or past knowledge instead.

In this case, such invariants typically capture that there is a particular relation (frequently capturing a post-condition P of a computation) between current knowledge and knowledge no older than the lag L. In the idealized case where all components have always up-to-date beliefs and their actions are instant the lag is equal to zero. In general, though, the lag is inversely proportional to the observed precision (assuming that precision depends on the oldness of observed data) and robustness (as in the case of real-time software control systems).

Definition 3. *(Present-past invariants) For a predicate P capturing the relation between valuation of knowledge elements $I_1, \ldots, I_n$ and $O_1, \ldots, O_m$, and the lag L, the expression $P^L_{p-p}[I_1, \ldots, I_n][O_1, \ldots, O_m]$ denotes the following* present-past invariant*:*

$$\forall t \in \mathbb{T}, \exists t_1, \ldots, t_n : 0 \leq t - t_i \leq L, i \in 1..n : \\ P(I_1[t_1], \ldots, I_n[t_n], O_1[t], \ldots, O_m[t])$$

In this context, we call $I_1, \ldots, I_n$ "input" variables and $O_1, \ldots, O_m$ "output" variables of the invariant so as to denote the correspondence of these variables to the inputs/outputs of the computation that is responsible for maintaining the invariant.

During refinement of a general invariant into (a conjunction of) present-past invariants, it is necessary to introduce assumptions 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 – e.g. assumption (4) in Figure 1.

Activity invariants. Another frequent form of timed invariants, used at a lower level of abstraction, closely reflects properties of a (soft) real-time activity while assuming read consistency with respect to the input knowledge of this activity, i.e., that each output knowledge valuation is based on the same or newer input knowledge valuation than the previous one. This is illustrated in Figure 2.

In this case, an *activity invariant* captures that the output knowledge valuation changes only as a result of performing the activity. Moreover, although reading the input knowledge of the activity, as well as computing and writing the output knowledge, takes some time, it never (altogether) exceeds the corresponding time limit (i.e., lag).

More rigorously, at any time the output knowledge valuation corresponds to the outcome of the activity applied on input knowledge valuation not older than the lag. Moreover, each output is based on same or newer inputs than the previous output.

Definition 4. *(Activity invariant) For a predicate P reflecting the post-condition of an activity with inputs $I_1, \ldots, I_n$ and outputs $O_1, \ldots, O_m$, and for lag L, the*

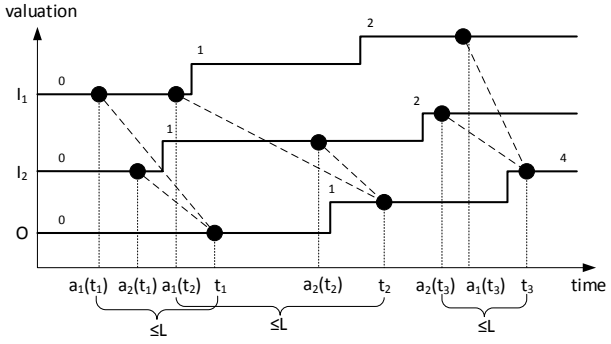


Fig. 2. Illustration of a valid knowledge valuation with respect to an activity where the output O represents sum of inputs I_1 and I_2, while meeting lag L.

expression $P^L_{act}[I_1, \ldots, I_n][O_1, \ldots, O_m]$ *denotes the following* activity invariant*:*

$$\exists a_1, \ldots, a_n : \mathbb{T} \to \mathbb{T}, \forall t \in \mathbb{T}, 0 \leq t - a_i(t) \leq L, a_i \textit{ non-decreasing}, i \in 1..n : \\ P(I_1[a_1(t)], \ldots, I_n[a_n(t)], O_1[t], \ldots, O_m[t])$$

where the non-decreasing function a_i *gives for each time* t *the corresponding time* t' *such that the valuation of* I_i *at* t' *was "used to compute" the valuation of* $O_1, \ldots, O_m$ *at* t*, as shown in Figure 2.*

Process invariants. At the lowest level of abstraction (i.e., in the leaves of the invariant decomposition), an activity invariant that captures local computation (i.e., with no distributed knowledge involved) while assuming read consistency is refined into an invariant capturing a periodic real-time component process – a *process invariant*.

Compared to activity invariants, process invariants introduce the additional constraint that the activity is performed exactly once in every *period*. The period thus becomes an elaboration of the activity lag, and the output knowledge evaluation is determined by the release time (time at which a task becomes ready for execution) and finish time in each period [3].

Specifically, such an invariant captures that if the current time is before the finish time of the process in the current period, then the outputs are the same as in the previous period (i.e., they correspond to the inputs used in the previous period). Otherwise, the outputs correspond to the inputs at the release time of the process in this period.

Definition 5. *(Process invariant) For a predicate* P *reflecting the post-condition of a periodic real-time process with inputs* $I_1, \ldots, I_n$*, outputs* $O_1, \ldots, O_m$*, and*

period L, the expression $P^{L}_{proc}[I_1, \ldots, I_n][O_1, \ldots, O_m]$ denotes the following process invariant*:*

$$\begin{gathered}\exists R, F : \mathbb{N} \to \mathbb{T} : E(x-1) \leq R(x) < F(x) < E(x)\ \forall x \in \mathbb{N}, \\ \forall p \in \mathbb{N}, \forall t \in \langle E(p-1), E(p)) : \\ t < F(p) \Rightarrow P(I_1[R(p-1)], \ldots, I_n[R(p-1)], O_1[t], \ldots, O_m[t]) \\ t \geq F(p) \Rightarrow P(I_1[R(p)], \ldots, I_n[R(p)], O_1[t], \ldots, O_m[t])\end{gathered}$$

where $E(n) : \mathbb{N}_0 \to \mathbb{T} = n \cdot L$, 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.

Note that unlike activity invariants, there is the same R for each I, reflecting that at the release time the process reads all the inputs atomically.

Exchange invariants. Similar to a process invariant, an activity invariant at the lowest level of abstraction that captures establishment of a belief (that can be addressed by ensemble knowledge exchange) while assuming distributed read consistency is refined into an invariant capturing periodic knowledge exchange of an ensemble – an *exchange invariant*.

Contrary to process invariants, exchange invariants assume that the input values might have been read at different times, since the inputs are potentially distributed (however, the times have to fit into the same period). Another difference is that exchange invariants consider also the knowledge propagation delays stemming e.g. from delays in data transfer over the network. An exchange invariant thus models a composite activity consisting of (i) knowledge transfer (with an upper bound on its duration), and (ii) periodic evaluation of the membership condition and knowledge exchange.

An important assumption is that each component executes the incoming knowledge exchange (i.e., knowledge exchange that updates the local component's knowledge) on its own, while the other components asynchronously send the required input knowledge. These composite activities may be partially overlapping to cater for situations where the knowledge transfer time is larger than the knowledge exchange period.

Definition 6. *(Exchange invariant) Let P be a predicate reflecting the post-condition of a periodic knowledge exchange with inputs $I_1, \ldots, I_n$, outputs $O_1, \ldots, O_m$, and period L. Provided that it takes at most T for the knowledge to become available at the component executing the knowledge exchange, the expression $P^{L,T}_{exc}[I_1, \ldots, I_n][O_1, \ldots, O_m]$ denotes the following* exchange invariant*:*

$$\begin{gathered}\exists a_1, \ldots, a_n : \mathbb{T} \to \mathbb{T}, \forall t \in \mathbb{T}, 0 \leq t - a_i(t) \leq T, a_i \text{ non-decreasing}, i \in 1..n : \\ \exists R, F : \mathbb{N} \to \mathbb{T} : E(x-1) \leq R(x) < F(x) < E(x)\ \forall x \in \mathbb{N}, \\ \forall p \in \mathbb{N}, \forall t \in \langle E(p-1), E(p)) : \\ t < F(p) \Rightarrow P(I_1[a_1(R(p-1))], \ldots, I_n[a_n(R(p-1))], O_1[t], \ldots, O_m[t]) \\ t \geq F(p) \Rightarrow P(I_1[a_1(R(p))], \ldots, I_n[a_n(R(p))], O_1[t], \ldots, O_m[t])\end{gathered}$$

where $E(n) : \mathbb{N}_0 \to \mathbb{T} = n \cdot L$, 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 gives for each time t the corresponding time t' such that the valuation of I_i that was available to the component executing the knowledge exchange at t was sent to the component at t'.

Note, that there is a (potentially) different a_i for each I_i, reflecting that the inputs can be sent to the component executing the knowledge exchange at different times. Moreover, there is the same t for each O_i, which corresponds to the assumption, that knowledge exchange is unidirectional, i.e., it writes only into the knowledge of one component, and thus the writes can be atomic.

3.1 Illustration of invariant patterns on the running example

Using the above-defined invariant patterns, the case-study invariants can be formalized as follows. Note that the patterns are not applicable for invariants 1 and 3, and are only partially applicable for invariant 4 (only for the left hand side of the implication), since 1 is a general invariant and 3 and 4 are assumptions.

(1) *The vehicle reaches its destination in time:*

$$\exists t \in \mathbb{T}, t \leq DEADLINE : v.pos[t] = DEST$$

(2) *The vehicle's plan is always based on CS data at most 6 minutes old:*

$$Plan_{p-p}^{6min}[t, v.pos, v.charge, CS_1, \ldots, CS_n][v.plan]$$

where the *Plan* predicate denotes the post-condition of the planning algorithm given the current time, current position, current charge, and CS data.

(3) *The vehicle's position is always in sync with the current plan:*

$$\forall t \in \mathbb{T} : v.pos[t] = v.plan[t](t)$$

(4) *When considering CS data no older than 10 minutes, the planner schedules reaching the destination in time.*

$$Plan_{p-p}^{10min}[t, v.pos, v.charge, CS_1, \ldots, CS_n][v.plan]$$
$$\Rightarrow \exists t' \in \mathbb{T}, t' \leq DEADLINE : v.plan[t](t') = DEST$$

(5) *The vehicle's plan is always computed from the local belief (over CS data) at most 2 minutes old.*

$$Plan_{act}^{2min}[t, v.pos, v.charge, v.belief][v.plan]$$

(6) *The belief of the vehicle over CS data is at most 4 seconds old.*

$$Belief_{p-p}^{4min}[CS_1, \ldots, CS_n][v.belief]$$

where the $Belief$ predicate denotes the condition of the vehicle's belief being equal to the CS data.

(7) *The vehicle computes the plan from the local belief (over CS data) periodically every 1 minute.*

$$Plan_{proc}^{1min}[t, v.pos, v.charge, v.belief][v.plan]$$

(8) *The vehicle updates its belief (over CS data) periodically every 2 minutes.*

$$Belief_{exc}^{2min}[CS_1, \ldots, CS_n][v.belief]$$

Naturally, the usage of invariant patterns particularly simplifies the lower-level, more technical invariants that capture computation activities. This allows for more concise and consistent invariant-based design.

4 Correctness by construction

A simplification of invariant-based design is not the only benefit of using the invariant patterns during invariant-based design. The main advantage is the ability of formal reasoning on the level of patterns instead of reasoning on the level of predicate logic upon knowledge valuations (since state-of-the-art theorem provers for such complex logics still do not have the necessary performance).

Thus, we propose a formal framework allowing for formal reasoning on the level of invariant patterns.

4.1 Basic pattern relations

First, we elaborate on the basic relations of the invariant patterns which correspond to the natural relations among the related software concepts of activity/activity with read consistency/process/ensemble.

A straightforward observation for a present-past invariant is that, given a particular knowledge valuation, if the outputs are always based on inputs within the given time limit, increasing the limit maintains this property. A similar observation holds for activity invariants. This is formalized in the following theorem.

Theorem 1. *(Maximal lag refinement) For $K \leq L$:*

$$P_{p-p}^{K}[I_1, \ldots, I_n][O_1, \ldots, O_m] \Rightarrow P_{p-p}^{L}[I_1, \ldots, I_n][O_1, \ldots, O_m]$$
$$P_{act}^{K}[I_1, \ldots, I_n][O_1, \ldots, O_m] \Rightarrow P_{act}^{L}[I_1, \ldots, I_n][O_1, \ldots, O_m]$$

Proof. A direct corollary of the lag/activity invariant definition. In particular, the existence of t_i such that $0 < t - t_i \leq K$ in $P_{p-p}^{K}[I_1, \ldots, I_n][O_1, \ldots, O_m]$ guarantees the existence of t_i such that $0 < t - t_i \leq L$ in $P_{p-p}^{L}[I_1, \ldots, I_n][O_1, \ldots, O_m]$ (similarly for a_i and $0 < x - a_i(x) \leq L$). □

One can also observe that the requirement of read consistency of inputs in addition to the time limit (in activity invariants) is a stronger requirement than the time limit only (in present-past invariants); this is formalized in the following theorem.

Theorem 2. *(Activity invariant implies present-past invariant) Assuming that* $I = I_1, \ldots, I_n$ *and* $O = O_1, \ldots, O_m$*, it holds:*

$$P^L_{act}[I][O] \Rightarrow P^L_{p-p}[I][O]$$

Proof. The existence of $t_1, \ldots, t_n$ for $P^L_{p-p}[I][O]$ is given by $a_1, \ldots, a_n$ of $P^L_{act}[I][O]$. In particular, $\forall t$ we set $t_i = a_i(t)$. □

A similar theorem can be formulated for the process and activity invariants. Here, the idea is that, in reality, a periodic process is actually a strict refinement of an activity with read consistency and time limit on input data. However, instead of considering the same time limit for both invariants as in previous cases, the activity invariant needs twice the time limit of the process invariant. This also 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, the period needs to be at most half of the desired response time [3]. For our invariant patterns, this fact is formalized in the following theorem.

Theorem 3. *(Process invariant implies activity invariant) Assuming that* $I = I_1, \ldots, I_n$ *and* $O = O_1, \ldots, O_m$*, it holds:*

$$P^L_{proc}[I][O] \Rightarrow P^{2L}_{act}[I][O]$$

Proof. Without loss of generality let us assume that $|I| = |O| = 1$. Given $t \in \mathbb{T}$ let $p = \lceil \frac{t}{L} \rceil$. The required $a : \mathbb{T} \to \mathbb{T}$ for $P^{2L}_{act}[I][O]$ is given by R and F from $P^L_{proc}[I][O]$ as follows:

$$a(t) = \begin{cases} R(p-1) & \text{if } t < F(p) \\ R(p) & \text{if } t \geq F(p) \end{cases}$$

First, we prove that $0 < t - a(t) \leq 2L$. Since $p = \lceil \frac{t}{L} \rceil$, then also $(p-1) \cdot L \leq t \leq p \cdot L$. According to Definition 5, $E(p-1) \leq R(p) < F(p) \leq E(p)$, where $E(p) = p \cdot L$. Therefore, given the properties of R, F, and $a(t)$, we have $E(p-2) \leq R(p-1) \leq a(t)$ and $a(t) < t$. Together, we have $(p-2) \cdot L \leq a(t) < t \leq p \cdot L$. Therefore, $0 < t - a(t) \leq 2L$.

Further, a is non-decreasing since R and F are non-decreasing. Thus, from $P^L_{proc}[I][O]$ we get $P^{2L}_{act}[I][O]$. □

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, as formulated by the following theorem.

Theorem 4. *(Exchange invariant implies activity invariant) Assuming that* $I = I_1, \ldots, I_n$ *and* $O = O_1, \ldots, O_m$*, it holds:*

$$P^{L,T}_{exc}[I][O] \Rightarrow P^{2L+T}_{act}[I][O]$$

Proof. The proof is similar to Theorem 3, differing only in the part relevant to knowledge transfer over network. For the purpose of the proof, we denote $R_i(p) = a_i(R(p)), \forall p \in \mathbb{N}$ for R and a_i from $P_{exc}^{L,T}[I][O]$.

Given $t \in \mathbb{T}$ let $p = \lceil \frac{t}{L} \rceil$. The required $a_i : \mathbb{T} \to \mathbb{T}$ for $P_{act}^{2L+T}[I][O]$ is given by R_i and F from $P_{exc}^{L,T}[I][O]$ as follows:

$$a_i : (t) = \begin{cases} R_i(p-1) & \text{if } t < F(p) \\ R_i(p) & \text{if } t \geq F(p) \end{cases}$$

First, we prove that $0 < t - a_i(t) \leq 2L + T$. Since $p = \lceil \frac{t}{L} \rceil$, then also $(p-1) \cdot L \leq t \leq p \cdot L$. According to Definition 6, $E(p-1) - T \leq R(p) - T \leq R_i(p) < F(p) \leq E(p)$, where $E(p) = p \cdot L$ (recall that $x - a_i^{ens}(x) \leq T$). Therefore, given the properties of R_i, F, and $a(t)$, we have $E(p-2) - T \leq R_i(p-1) \leq a(t)$ and $a(t) < t$. Together, we have $(p-2) \cdot L - T \leq a(t) < t \leq p \cdot L$. Therefore, $0 < t - a(t) \leq 2L + T$.

Further, a_i is non-decreasing since R_i and F are non-decreasing. Thus, from $P_{exc}^{L,T}[I][O]$ we get $P_{act}^{2L+T}[I][O]$. □

4.2 Pipeline decomposition

Here, we present a logical framework that would enable for formal reasoning about refinement in a particular form of decomposition – *pipeline decomposition*, which due to its relative generality covers most practical cases of invariant decomposition. Specifically, we focus on the level of activity invariants, as they represent a suitable level of abstraction, generalizing both process and exchange invariants.

As an important observation, the fact that a decomposition is actually a refinement of the parent invariant is, with respect to time, largely affected by sharing of invariant variables among the child invariants. Thus, we introduce the concept of *dependency chain*. A vector of activity invariants forms a dependency chain if some of the output variables of a invariant in the vector are among the input variables of the next invariant in the vector. This is formalized in the following definition.

For brevity, we introduce the following notation. Given an activity (or process/exchange) invariant $P_{act}^{L}[I_1, \ldots, I_n][O_1, \ldots, O_m]$, $In(P)$ denotes the set $\{I_1, \ldots, I_n\}$, while $Out(P)$ denotes the set $\{O_1, \ldots, O_m\}$.

Definition 7. *(Dependency chain) Each vector* $\left(P_{1\,act}^{L_1}, \ldots, P_{p\,act}^{L_p}\right)$ *of invariants forms a* dependency chain *iff:*

$$\forall i \in \{1, \ldots, p-1\}\, \exists O, I : \\ O \in Out(P_i) \wedge I \in In(P_{i+1}) \wedge O = I$$

In a pipeline decomposition the children reflect simple pipeline-like flows among the corresponding activities that refine the parent activity. A formal interpretation is given in the following definition.

Definition 8. *(Pipeline decomposition) Having a parent invariant* P^{L}_{act}*, a set of child invariants* $\left\{P^{L_i}_{i\,act}, i = 1..p\right\}$ *forms a* pipeline decomposition *of* P^{L}_{act} *iff:*

(i) each input variable of the parent is an input variable of exactly one child:

$$\forall I \in In(P)\ \exists! j \in \{1,\ldots,p\} : I \in In(P_j)$$

(ii) each output variable of the parent is an output variable of exactly one child:

$$\forall O \in Out(P)\ \exists! j \in \{1,\ldots,p\} : O \in Out(P_j)$$

(iii) the decomposition includes only such dependency chains, in which (a) all input variables of the first invariant are input variables of the parent, (b) all output variables of the last invariant are output variables of the parent, (c) for each two consecutive invariants within the dependency chain, the output variables of the former are exactly the input variables of the latter:

$$\forall \mathcal{C} = \left(P^{L_{i_1}}_{i_1\,act}, \ldots, P^{L_{i_q}}_{i_q\,act}\right), \{i_1,\ldots,i_q\} \subseteq \{1,\ldots,p\}, \mathcal{C}\ \textit{dependency chain:}$$
$$In(P_{i_1}) \subseteq In(P) \wedge Out(P_{i_q}) \subseteq Out(P)$$
$$\wedge\ \forall j = i_1..i_{q-1}\ Out(P_j) = In(P_{j+1})$$

(iv) the decomposition includes only such dependency chains that do not share input/output variables:

$$\forall \mathcal{C}_1 = \left(P^{L_{i_1}}_{i_1\,act}, \ldots, P^{L_{i_q}}_{i_q\,act}\right), \{i_1,\ldots,i_q\} \subseteq \{1,\ldots,p\}, \mathcal{C}_1\ \textit{dependency chain},$$
$$\forall \mathcal{C}_2 = \left(P^{L_{j_1}}_{j_1\,act}, \ldots, P^{L_{j_r}}_{j_r\,act}\right), \{j_1,\ldots,j_r\} \subseteq \{1,\ldots,p\}, \mathcal{C}_2\ \textit{dependency chain},$$
$$\forall P^{L_k}_{k\,act} \in \mathcal{C}_1, \forall P^{L_l}_{l\,act} \in \mathcal{C}_2 :$$
$$\mathcal{C}_1 \neq \mathcal{C}_2 \Rightarrow \left(In(P^{L_k}_{k\,act}) \cup Out(P^{L_k}_{k\,act})\right) \cap \left(In(P^{L_l}_{l\,act}) \cup Out(P^{L_l}_{l\,act})\right) = \emptyset$$

An example is the decomposition of (2) into (5) and (6) in the running example.

Intuitively, the definition of pipeline decomposition requires the children to reflect simple parallel pipeline-like flows (dependency chains) among the corresponding activities that refine the parent activity.

For pipeline decomposition, a straightforward rule for determining refinement can be formulated. In a correct refinement, provided that the decomposition is logically consistent with the parent invariant when not considering time, the lag of the parent invariant should be at least the sum of the lags of the invariants in the longest (in terms of time) pipeline (i.e., dependency chain) of the decomposition. Indeed, this intuitive observation was confirmed in our invariant-based formalism as demonstrated in the following theorem.

Theorem 5. *(Activity invariant pipeline refinement) Having invariant* P^{L}_{act} $[I_1,\ldots,I_n][O_1,\ldots,O_m]$ *and its pipeline decomposition* $\mathcal{D} = \left\{P_1{}^{L_1}_{act}, \ldots, P^{L_p}_{p\,act}\right\}$*, the decomposition is a refinement of the parent, i.e., it holds that* $P_1{}^{L_1}_{act} \wedge \cdots \wedge P^{L_p}_{p\,act} \Rightarrow P^{L}_{act}$*, if:*

(i) $P_1 \wedge \cdots \wedge P_p \Rightarrow P$, *i.e., the decomposition is logically consistent without considering time*

(ii) for each dependency chain $\mathcal{C} = \left(P_{i_1\,act}^{\;L_{i_1}}, \ldots, P_{i_q\,act}^{\;L_{i_q}}\right)$ *in* $\mathcal{D}$ *it holds that* $\sum_{j=i_1}^{i_q} L_j \leq L$, *i.e., the lag of the parent invariant is at least the sum of the lags of the longest (in terms of time) dependency chain among the child invariants.*

Proof. To prove the above theorem, we need to prove that given $\mathcal{D}$, P, and the assumptions (i) and (ii), the following lemma holds:

$$P_{1\,act}^{\;L_1} \wedge \cdots \wedge P_{p\,act}^{\;L_p} \Rightarrow (P_1 \wedge \cdots \wedge P_p)_{act}^{L}$$

Then, the correctness of the theorem is an immediate result of this lemma and the assumption (i). To prove the lemma, let $Q_{act}^{L} \stackrel{\text{def}}{=} (P_1 \wedge \cdots \wedge P_p)_{act}^{L}$.

Without loss of generality, let us assume that each dependency chain $\mathcal{C} = \left(P_{i_1\,act}^{\;L_{i_1}}, \ldots, P_{i_q\,act}^{\;L_{i_q}}\right)$ in $\mathcal{D}$, its first invariant $P_{i_1\,act}^{\;L_{i_1}}$ in particular, has only one input variable (i.e., $I_{\mathcal{C}}$). Also, let us assume that $\mathcal{C}$, its last invariant $P_{i_q\,act}^{\;L_{i_q}}$ in particular, has only one output variable (i.e., $O_{\mathcal{C}}$). Similarly, we assume that all the intermediate invariants within $\mathcal{C}$ have exactly one input and one output variable. This assumption is safe since the multiple input/output variables can be merged into one as they are referred exactly from one other invariant (which is also in $\mathcal{C}$).

For the variable $I_{\mathcal{C}}$, we define the $a_{\mathcal{C}} : \mathbb{T} \rightarrow \mathbb{T}$ required for Q_{act}^{L} (according to the Definition 4) as follows:

$$a_{\mathcal{C}}(t) \stackrel{\text{def}}{=} a_{i_1}\left(a_{i_2}\left(\ldots a_{i_q}\left(t\right)\ldots\right)\right)$$

where $a_{i_1}, \ldots, a_{i_q}$ are taken from to $P_{i_1\,act}^{\;L_{i_1}}, \ldots, P_{i_q\,act}^{\;L_{i_q}}$.

Because $\sum_{j=i_1}^{i_q} L_j \leq L$ and $0 < x - a_{i_1}(x) \leq L_{i_1}, \ldots, 0 < x - a_{i_q}(x) \leq L_{i_q}$, it holds that $0 < x - a_{\mathcal{C}} \leq L$.

The assumption of the above lemma (i.e., $P_{1\,act}^{\;L_1} \wedge \cdots \wedge P_{p\,act}^{\;L_p}$) and the properties of the dependency chain $\mathcal{C} = \left(P_{i_1\,act}^{\;L_{i_1}}, \ldots, P_{i_q\,act}^{\;L_{i_q}}\right)$ as a part of the pipeline decomposition $\mathcal{D}$ give us the following corollary:

$$\begin{gathered}
P_{i_1}(I_{\mathcal{C}}[a_{i_1}(a_{i_2}(\ldots a_{i_q}(t)\ldots))], O_{i_1}[a_{i_2}(\ldots a_{i_q}(t)\ldots)]) \wedge O_{i_1} = I_{i_2} \wedge \\
P_{i_2}(I_{i_2}[a_{i_2}(a_{i_3}(\ldots a_{i_q}(t)\ldots))], O_{i_2}[a_{i_3}(\ldots a_{i_q}(t)\ldots)]) \wedge O_{i_2} = I_{i_3} \wedge \\
\vdots \\
P_{i_q}(I_{i_q}[a_{i_q}(t)], O_{\mathcal{C}}[t])
\end{gathered}$$

By combining these corollaries for each dependency chain in the pipeline decomposition $\mathcal{D}$ of Q (i.e., each input and output variable of Q), we get:

$$Q\left(I_1\left[a_1\left(t\right)\right], \ldots, I_n\left[a_n\left(t\right)\right], O_1\left[t\right], \ldots O_n\left[t\right]\right)$$

where I_i, O_i, and a_i correspond to the dependency chain $\mathcal{C}_i$ in $\mathcal{D}$.

By combining all the above facts, we get: $P_{1\,act}^{\;L_1} \wedge \cdots \wedge P_{p\,act}^{\;L_p} \Rightarrow Q_{act}^{L}$

□

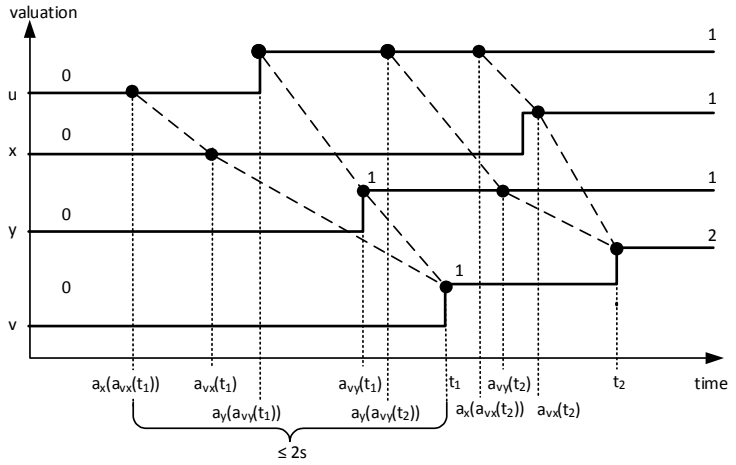


Fig. 3. A counterexample illustrating the importance of the pipeline refinement assumption in Theorem 5.

4.3 More complex types of refinement

The assumption of pipeline decomposition in Theorem 5 is essential for its correctness. This means that in the case of a decomposition that does not respect all four points of Definition 8, applying Theorem 5 can lead to the wrong results. To support this claim and highlight the importance of strictly following the above-mentioned definition, we present the following counterexample to the relaxed Theorem 5 (where the assumption of pipeline decomposition is lifted).

Counterexample to relaxed Theorem 5. Consider the parent invariant $P_p \stackrel{\text{def}}{=} (v = 2u)^{2s}_{act}[u][v]$, that is decomposed into three sub-invariants:

$$P_\alpha \stackrel{\text{def}}{=} (x = u)^{1s}_{act}[u][x], \quad P_\beta \stackrel{\text{def}}{=} (y = u)^{1s}_{act}[u][y], \quad P_\gamma \stackrel{\text{def}}{=} (v = x + y)^{1s}_{act}[x, y][v].$$

This decomposition is not a pipeline decomposition, because the input variable of the parent (variable u) is input of more than one children in the decomposition (both P_α and P_β), thus invalidating the first point of Definition 8. The relaxed Theorem 5 would ensure that this decomposition is a refinement. However, if we consider the trace illustrated in Figure 3, it is obvious that although the trace is valid for all the sub-invariants P_α, P_β, and P_γ, it is not valid for the parent invariant P_p, as there cannot be an $a_p(t)$ such that $v[t_1] = 1 = 2 * u[a_p(t_1)]$. □

The reason why the relaxed Theorem 5 does not work for the counterexample is that while the parent works with the valuation of a at a single time instant, the decomposition employs the valuation of a at two different time instants (by

aliasing to x and y). This observation applies in general. Moreover, for some decompositions it appears that it is not possible to formulate similar theorems.

5 Discussion and Conclusions

The choice of the proposed formalization of invariants and invariant patterns in higher-order predicate logic was driven by the practical reason of being able to formulate and prove the relevant theorems that hold in different invariant refinements. Other forms of formalization would have been more appropriate when different goals are pursued by the formalization task. For example, the use of a real-time temporal logic [12] would have been a sensible choice if we would like to use IRM model fragments as input for model-checking purposes.

Indeed, formalization of goals in goal models in real-time LTL has already been pursued in the context of both KAOS [13] and Tropos [6] (e.g., Formal Tropos [5]), two of the most prominent requirements engineering frameworks. Our invariant refinement patterns can be compared to the goal refinement patterns *à* la KAOS [4], which encode known refinement tactics. The difference is that KAOS patterns can be formally checked with a theorem prover, while our patterns have to be manually proven, as state-of-the-art theorem provers cannot cope with the complexity of our expressive logic.

The invariant decomposition in IRM is inspired by the decomposition of system-level goals into sub-goals, assumptions and domain properties in KAOS. A similar approach is also pursued within Tropos, where goals, soft-goals, tasks, and dependencies and identified and iteratively decomposed from the perspective of the individual agents. The differences lie in that (i) neither KAOS nor Tropos provide a direct translation to the implementation-level concepts of autonomic components and ensembles; (ii) the objective of IRM is not to produce requirements documents (like KAOS), but software architectures; (iii) IRM invariants do not focus on future states (like goals in Tropos), but on knowledge valuation at every time instant, fitting better the design of feedback-based systems.

The diagrams used to illustrate the knowledge valuation in time in IRM (e.g., Fig. 2 and 3) are reminiscent of timed UML 2 interaction diagrams [11], as they capture the system behavior over time in a declarative way. However, UML 2 activity diagrams focus on the message exchange between predefined instances, whereas IRM invariants capture the evolution in the knowledge of distributed components (which could be implemented by exchange of messages among them) that is necessary in order for certain system-level requirements to be met.

To conclude, in this paper we have provided a formal framework for invariant refinement in the context of the Invariant Refinement Method (IRM). Our approach is modeling the invariants in higher-order predicate logic and identifying common invariant types (patterns) at different levels of abstraction. Some of the refinement relations between different patterns have also been formally proven (via mathematical theorems): present-past to activity invariants, activity to process/exchange invariants, and pipeline decomposition of activity/process/exchange invariants. More complex types of refinement have to be

investigated separately in order to be able to formulate similar theorems. This is the focus of our future work.

Another element of future work is to test the proposed design method in a real-scale case study with real system designers.

Acknowledgements. This work was partially supported by the EU project ASCENS 257414 and by Charles University institutional funding SVV-2014-260100. The research leading to these results has received funding from the European Union Seventh Framework Programme FP7-PEOPLE-2010-ITN under grant agreement n°264840.

References

1. Abeywickrama, D.B., Bicocchi, N., Zambonelli, F.: SOTA: Towards a General Model for Self-Adaptive Systems. In: Proc. of WETICE. pp. 48–53. IEEE (2012)
2. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: DEECo – an Ensemble-Based Component System. In: Proc. of CBSE'13, Vancouver, Canada. pp. 81–90. ACM (Jun 2013)
3. Buttazzo, G.C.: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Springer, 3rd edn. (2011)
4. Darimont, R., van Lamsweerde, A.: Formal Refinement Patterns for Goal-Driven Requirements Elaboration. In: Proceedings of FSE'96. pp. 179–190. ACM (1996)
5. Fuxman, A., Pistore, M., Mylopoulos, J., Traverso, P.: Model Checking Early Requirements Specifications in Tropos. In: Proc. of RE'01, Toronto, ON, Canada. pp. 174–181. IEEE (Aug 2001)
6. Giorgini, P., Kolp, M., Mylopoulos, J., Pistore, M.: The Tropos Methodology: An Overview. In: Methodologies And Software Engineering For Agent Systems, pp. 89–106. Kluwer Academic Publishers (2004)
7. Hölz, M., Wirsing, M.: Towards a System Model for Ensembles. In: Formal modeling, pp. 241–261. Springer-Verlag (2012)
8. Hölzl, M., et al.: Engineering Ensembles: A White Paper of the ASCENS Project. ASCENS Deliverable JD1.1 (2011), Online: http://www.ascens-ist.eu/whitepapers
9. Hölzl, M., Rauschmayer, A., Wirsing, M.: Software engineering for ensembles. In: Software-Intensive Systems and New Computing Paradigms, pp. 45–63. Springer-Verlag (2008)
10. Keznikl, J., Bures, T., Plasil, F., Gerostathopoulos, I., Hnetynka, P., Hoch, N.: Design of Ensemble-Based Component Systems by Invariant Refinement. In: Proc. of CBSE'13, Vancouver, Canada. pp. 91–100. ACM (Jun 2013)
11. Knapp, A., Störrle, H.: Efficient Representation of Timed UML 2 Interactions. In: Amyot, D., Fonseca i Casas, P., Mussbacher, G. (eds.) System Analysis and Modeling: Models and Reusability, LNCS, vol. 8769, pp. 110–125. Springer (2014)
12. Koymans, R.: Specifying Message Passing and Time-Critical Systems with Temporal Logic. v. 651 of LNCS, Springer-Verlag (1992)
13. Lamsweerde, A.V.: Requirements engineering in the year 00: a research perspective. In: Proceedings of ICSE'00, Limerick, Ireland. pp. 5–19. ACM (Jun 2000)
14. Serbedzija, N., Reiter, S., Ahrens, M., Velasco, J., Pinciroli, C., Hoch, N., Werther, B.: Requirement Specification and Scenario Description of the ASCENS Case Studies. Deliverable D7.1 (2011), available online: http://www.ascens-ist.eu/deliverables

4.5 Model-Driven Design of Ensemble-Based Component Systems

Ilias Gerostathopoulos

In Joint Proceedings of the **ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014) Poster Session and the ACM Student Research Competition**.

Published electronically by CEUR Workshop Proceedings,
volume 1258,
pages 63-68,
September 2014.

The original version is available electronically from the publisher's site at http://ceur-ws.org/Vol-1258/.

Summary of the Paper

This paper, published as [Ger14], serves as an overview of the model-driven design and development process proposed in this thesis. It has the form of a research abstract for the ACM Student Research Competition at MODELS 2014 conference. Following the respective call for participation [22], the paper describes the (i) research problem and motivation, (ii) background and related work, (iii) approach and uniqueness, (iv) results, and (v) contributions. As such, it provides a preliminary account on the different elements of the work described in detail in this thesis.

At the same time, the paper introduces an extension of the Invariant Refinement Method (IRM) to account for alternative invariant decompositions, and sketches its application at design time and at runtime. The extension, called *IRM for Self-Adaptation* (IRM-SA) was originally described in [BGH+14b], and refined in the technical report included in Section 4.6. The main idea of IRM-SA is to accommodate different invariant-based designs in the same invariant model (graph) by allowing invariants to be OR-decomposed – in the work so far, invariants could only be AND-decomposed. This supports designing for self-adaptivity in software-intensive Cyber-Physical Systems modelled according to the ensemble paradigm (called *ensemble-based component systems* in this paper), as different configurations corresponding to alternative invariant decompositions can be selected at runtime.

The paper provides a first version of the IRM-SA *design process*, comprising the steps to be followed in order to obtain a complete design represented by an IRM-SA model of the system-to-be. The process is backed up by a graphical editor and a model-driven toolchain capable of producing jDEECo code artifacts (Section 5.2.1). A refined version of the IRM-SA design process is described in the technical report of Section 4.6.

Moreover, the paper introduces the idea of using the IRM-SA model at runtime to provide self-adaptation via switching between different configurations prescribed by alternative invariant decompositions. The initial efforts for implementing such a functionality in jDEECo (which led to the jDEECo IRM plugin described in Section 5.2.2) are also reported.

Comments on Authorship

Although the main ideas of the paper were formed in collaboration with my supervisors, the single authorship of this paper reflects my individual participation in the ACM Student Research Competition at MODELS 2014 conference, where I received the 2nd place Graduate level award.

Model-Driven Design of Ensemble-Based Component Systems

Ilias Gerostathopoulos

Faculty of Mathematics and Physics
Charles University in Prague
Malostranske Namesti 25, 11800, Prague, Czech Republic
iliasg@d3s.mff.cuni.cz

Abstract. In this research abstract we describe our approach towards the design of ensemble-based component systems. Our motivation lies in the fact that, in these systems, tracing the behavior of individual constituents to system-level goals and requirements is challenging. Our approach is based on a novel invariant-based model that achieves the desired traceability. Along with using the model in a method that allows for systematic contractual design, we employ the model at runtime to achieve dynamic adaptation on the basis of requirements reflection.

Keywords: ensembles, invariants, system design, traceability

1 Introduction

In the beginning, things were not going well. The heavy storm had damaged the network infrastructure so heavily that temperature and moisture sensors on the tarmac could not communicate with their base stations any longer. This meant that continuous analysis of tarmac condition had to stop until the network cables were back in place and sensors started providing fresh measurements to the base stations. In face of the danger of failing in their task to disseminate the sensed data, the sensors switched to ad-hoc communication mode: they propagated their data to software modules inside the vehicles heading towards the base stations; the vehicles acted as network relays for the ad-hoc network and "augmented" sensors for the base stations. Even with considerable delays compared to the default mode, the system managed to keep a sufficient level of operation stability.

Although developing a *software-intensive cyber-physical system* (siCPS) [12] such as the above road sensing system is already technically feasible, there are challenges related to streamlining the design and development of such systems.

DEECo component model [1,4] has been proposed within the ASCENS FP7 project [11] as a modeling approach suitable for the development of siCPS. A DEECo application consists of a number of components and interaction templates, based on which dynamic component groups – *ensembles* [11] – are established at runtime. A DEECo component comprises state (referred to as *knowledge*) and *processes* which periodically read and/or update its knowledge, similar

```
1 ensemble PropagateTemperatureToVehicles:
2  coordinator: TemperatureSensor
3  member: Vehicle
4  membership:
5   distance(coordinator.position, member.position) < THRESHOLD
6  exchange:
7   member.temperatureMap ←(coordinator.id, coordinator.temperature)
8  scheduling: periodic( 15 secs )
```

Fig. 1. Example of a DEECo ensemble definition in the road sensing system.

to processes in a real-time system. Interaction is allowed only between components within an ensemble and takes the form of knowledge exchange. An ensemble definition (Fig. 1) specifies (i) a *membership* condition, i.e., under which condition (evaluated on components' knowledge) one *coordinator* and potentially many *member* components should interact, and (ii) an *exchange* function, i.e., which knowledge exchange should be performed within the established group. We view DEECo as an instantiation of the new class of *ensemble-based component systems* (EBCS), and use it to demonstrate our EBCS design approach.

The **problem** in EBCS is that it is difficult to associate the low-level concepts of periodic computation and conditional knowledge exchange to system-level goals and requirements applicable in different operational contexts. This problem manifests itself both at design time and at runtime. At design time the challenge is: *"How to **design** an ensemble-based system so that its situation-specific system-level goals are consistently mapped to implementation-level artifacts?"*; at runtime the challenge becomes: *"How to **trace** the runtime behavior to situation-specific system-level goals to achieve runtime compliance checking?"*.

The **objective** of this research is thus to investigate the *design dimension* of ECBS and propose a model that provides *dependability* (in the form of traceability to system-level goals) and *adaptability* (in the form of adjusting to different operational contexts/situations). We aim for using the model both to guide the design of EBCS (Sect. 2.1), and to achieve runtime compliance checking and model-based adaptation (Sect. 2.2).

2 Approach: Invariant-Based Model

Our approach is based on the observation that component processes and knowledge exchange activities in EBCS are feedback loops that constantly maintain the property of being within the bounds of normal operation – *operational normalcy*. We have thus proposed the *invariant* concept to model the operational normalcy at every time instant [13]. Syntactically, an invariant is an expression that relates the input and output knowledge of an (abstract) activity, e.g. *"Vehicle's V belief over sensor S::temperature – V::temperatureMap – is updated every 30 secs."*. A key assumption here is that system-level goals in EBCS can also be described declaratively and thus modeled with the invariant construct. For instance, one such high-level invariant in our running example is *"Temperature readings must reach the base stations within 120 secs"*.

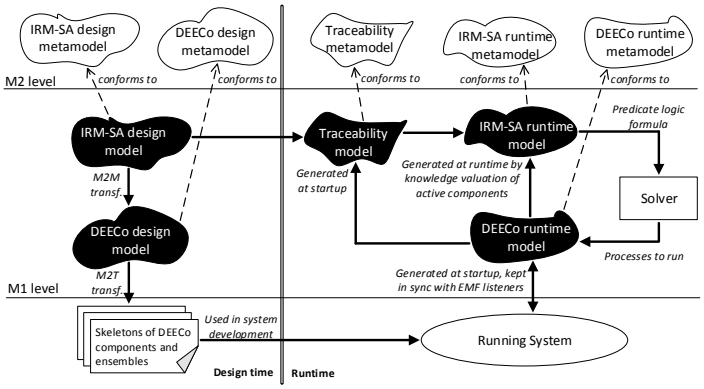


Fig. 2. Overview of the IRM-SA approach.

Armed with the invariant concept, we have proposed the *Invariant-Refinement Method for Self-Adaptation – IRM-SA* [5,13], whose goal is to link high-level invariants (corresponding to system-level goals) to low-level ones (corresponding to concrete activities of the software system). The output of the method is the *IRM-SA design model*; this model can be used (i) to generate DEECo code skeletons via the series of model transformations depicted on the left part of Fig. 2 and (ii) to enable online checking of invariant satisfaction and system adaptation via a models@runtime approach (illustrated on the right part of Fig. 2).

2.1 Design with IRM-SA

In this section we present the design process of IRM-SA. It is a mixed top-down/bottom-up iterative process where invariants are refined into sub-invariants by means of decomposition (e.g. structural elaboration). The process comprises:

1. Identification of the top-level goals and specification of top level invariants of the system-to-be, e.g. invariant [i1] in Fig. 3.
2. Identification of the *design components* by asking "which knowledge does each invariant involve and where is this knowledge obtained from?". At the design stage, a component is a participant/actor of the system-to-be, comprising internal state. In our example, the identified components are the `TemperatureSensor`, `BaseStation`, and `Vehicle`.
3. Decomposition of each non-leaf invariant by asking "**how** can this invariant be satisfied?". Leaf invariants are either *process invariants* (e.g. invariant [p1]) or *exchange invariants* (e.g. invariant [e2]) that can be mapped 1-to-1 to component processes or ensemble definitions, respectively. For instance, the exchange invariant [e2] can be mapped to the `PropagateTemperatureToVehicles` ensemble of Fig. 1.
4. Identification of any other activities that have to be performed in the system and specification of invariants out of them (not demonstrated here).

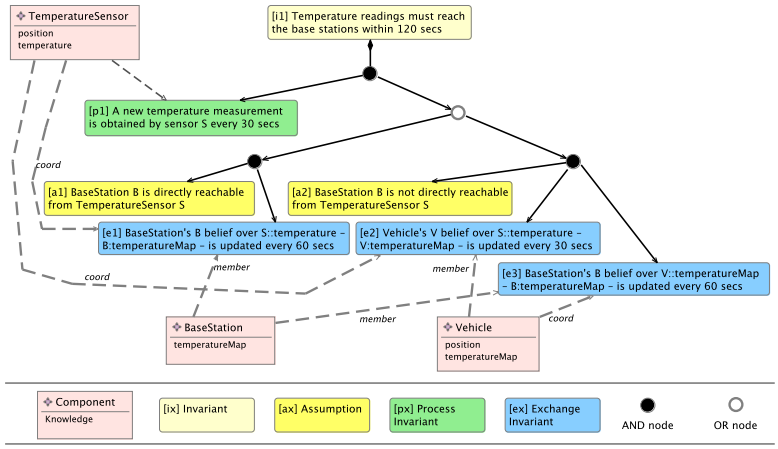


Fig. 3. Example of an IRM-SA design model.

5. Composition of dangling invariants together by asking "**why** do we need to satisfy these invariants?". This step is also not demonstrated in our example.
6. Capturing of the situation that conditions every *situation-specific invariant* using *assumptions* (e.g. invariant [a1]). An assumption is a special type of invariant that is expected to be maintained by the environment.
7. Identification of alternative (OR) decompositions according to the different situations identified at step 6. In our example, the right-most part of the top-level decomposition is OR-decomposed to capture the fact that different invariants should hold when a `BaseStation` is out of direct reach.

The IRM-SA design process is backed up by a prototype design tool (used to produce the IRM-SA model of Fig. 3) and a Java code generation tool, based on Eclipse's EMF and Epsilon toolchains; both are accessible via `http://d3s.mff.cuni.cz/projects/components_and_services/irm-sa/`.

2.2 Runtime compliance checking and adaptation

To check which invariants hold at runtime and adapt the system accordingly, we follow a models@runtime approach [17]. As a first step, the running system is reflected into an architectural model (*DEECo runtime model* in Fig. 2) that captures the running component processes and established ensembles. Along with a *traceability model*, which contains the mapping between design and runtime artifacts, DEECo runtime model is used to generate another model that captures the runtime state at the requirements level (*IRM-SA runtime model*). This is basically an instantiation of the IRM-SA design model in which design components are mapped to concrete component instances and invariants are associated with boolean values. This is done by associating the invariants and the

computable assumptions to *monitors* (implemented as Boolean methods in Java) that evaluate the condition under which each invariant is considered to hold.

The second step involves reasoning on the generated IRM-SA runtime model. As an illustration of one possible way to do this, we are translating the model into a predicate logic formula which can be automatically evaluated by a solver (we use Sat4j [16]). The result of the solver is then used to enact changes on the DEECo runtime model (currently by starting/stopping processes corresponding to invariants chosen in the OR-decompositions), which are eventually propagated to the running system, as illustrated on the right-most part of Fig. 2.

A proof-of-concept implementation of IRM-SA-based adaptation is accessible via `http://d3s.mff.cuni.cz/projects/components_and_services/irm-sa/`.

On-going work. We are currently investigating (i) the fuzzification of invariant evaluation to achieve more fine-grained control, and (ii) more elaborate adaptation actions (e.g. changing a component's period at runtime). To evaluate our approach we are conducting experiments to measure the applicability of our adaptation loop in practical settings (e.g. in face of frequent component disconnections). We have also designed and conducted a pilot of a controlled experiment (empirical study) to evaluate the effectiveness of the IRM-SA process.

3 Related Work

Systematic elaboration of requirements has been advocated by goal-oriented requirements engineering approaches, such as KAOS [7,15] and Tropos [3,9]. Although we draw inspiration from them we differentiate in the following [8]: (i) neither KAOS nor Tropos are tailored for ensemble-based systems, whereas IRM-SA provides a direct translation to the implementation-level concepts of autonomous components and ensembles; (ii) compared to KAOS, the objective of the IRM-SA method is not to create requirements documents (e.g., SRS), but software architectures; (iii) compared to Tropos, which also supports design of dynamic systems, IRM-SA concepts (i.e., invariants) do not focus on future states (like goals in Tropos), but on knowledge evaluation at every time instant, fitting better the design of feedback loop-based systems.

Our approach towards adaptation fits into the conceptual model proposed by Kramer and Magee [14], where the IRM-SA model stands as a domain-specific goal management layer. Our adaptation mechanism also follows the proposals for explicit representation of requirements as runtime entities [2,6].

Compositional definition of architecture configurations based on individual variation points and runtime reconfiguration is also employed in Dynamic Software Product Lines [10]. Our main difference is that, in IRM-SA, decomposition carries the formal semantics of refinement, i.e., in an AND (resp. OR) decomposition the conjunction (resp. disjunction) of the children entails the parent.

Acknowledgments. The research leading to these results has received funding from the European Union Seventh Framework Programme FP7-PEOPLE-2010-ITN under grant agreement n°264840.

References

1. Al Ali, R., Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: DEECo: An Ecosystem for Cyber-Physical Systems. In: Companion Proc. of ICSE'14, Hyderabad, India. pp. 610–611. ACM (Jun 2014)
2. Bencomo, N., Whittle, J., Sawyer, P., Finkelstein, A., Letier, E.: Requirements Reflection: Requirements as Runtime Entities. In: Proc. of ICSE '10, Cape Town, South Africa. pp. 199–202. ACM (2010)
3. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An Agent-Oriented Software Development Methodology. Autonomous Agents and Multi-Agent Systems 8(3), 203–236 (May 2004)
4. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: DEECo – an Ensemble-Based Component System. In: Proc. of CBSE'13, Vancouver, Canada. pp. 81–90. ACM (Jun 2013)
5. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F., Plouzeau, N.: Adaptation in Cyber-Physical Systems: from System Goals to Architecture Configurations. Tech. rep., D3S-TR-2014-01, Charles University (Jan 2014), http://d3s.mff.cuni.cz/publications/download/D3S-TR-2014-01.pdf
6. Cheng, B., et al.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems, LNCS, vol. 5525, pp. 1–26. Springer Berlin Heidelberg (2009)
7. Dardenne, A., Van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Science of Computer Programming 20(April), 3–50 (1993)
8. Gerostathopoulos, I., Bures, T., Hnetynka, P.: Position Paper: Towards a Requirements-Driven Design of Ensemble-Based Component Systems. In: Proc. of HotTopiCS workshop of ICPE'13. pp. 79–86. ACM (Apr 2013)
9. Giorgini, P., Kolp, M., Mylopoulos, J., Pistore, M.: The Tropos Methodology: An Overview. In: Methodologies and Software Engineering for Agent Systems, pp. 89–106. Kluwer Academic Publishers (2004)
10. Hinchey, M., Park, S., Schmid, K.: Building Dynamic Software Product Lines. Computer 45(10), 22–26 (Oct 2012)
11. Hölzl, M., et al.: Engineering Ensembles: A White Paper of the ASCENS Project. ASCENS Deliverable JD1.1 (2011), Online: http://www.ascens-ist.eu/whitepapers
12. 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 and New Computing Paradigms, LNCS, vol. 5380, pp. 1–44. Springer Berlin Heidelberg (2008)
13. Keznikl, J., Bures, T., Plasil, F., Gerostathopoulos, I., Hnetynka, P., Hoch, N.: Design of Ensemble-Based Component Systems by Invariant Refinement. In: Proc. of CBSE'13, Vancouver, Canada. pp. 91–100. ACM (Jun 2013)
14. Kramer, J., Magee, J.: A Rigorous Architectural Approach to Adaptive Software Engineering. Journal of Computer Science and Technology 24(2), 183–188 (2009)
15. Lamsweerde, A.V., Darimont, R., Letier, E.: Managing Conflicts in Goal-Driven Requirements Engineering. IEEE Trans. on Soft. Engin. 24(11), 908–926 (1998)
16. Le Berre, D., Parrain, A.: The Sat4j library, release 2.2. Boolean Modeling and Computation 7, 59–64 (2010)
17. Morin, B., Barais, O., Jezequel, J.M., Fleurey, F., Solberg, A.: Models at Runtime to Support Dynamic Adaptation. Computer 42(10), 44–51 (2009)

4.6 Self-Adaptation in Cyber-Physical Systems: from System Goals to Architecture Configurations

Ilias Gerostathopoulos,
Tomáš Bureš,
Petr Hnětynka,
Jaroslav Keznikl,
Michal Kit,
Frantisek Plášil,
Noël Plouzeau

Tech. Report No. D3S-TR-2015-02,
Department of Distributed and Dependable Systems,
Charles University in Prague,
April 2015.

Available online on the department's site: http://d3s.mff.cuni.cz/publications/download/D3S-TR-2015-02.pdf

An article based on this technical report has been submitted to a journal and is presently at major review stage.

Summary of the Paper

This technical report, available as [GBH+15b], provides the second main contribution of this thesis, i.e. the extension of the Invariant Refinement Method (IRM) to allow design for self-adaptivity. As such, it directly addresses the objective **O3** (Section 2.3). The focus is on the domain of "self-adaptive Cyber-Physical Systems (CPS) with operating in dynamic environments", which coincides with the domain of software-intensive CPS (siCPS) that is the focus of this thesis.

The main idea of the report is to tackle the challenge of designing siCPS that are both dependable and self-adaptive, and can cope with situations of high *operational uncertainty*, i.e. uncertainty related to the infrastructure of siCPS and leads to temporary network disconnections, hardware failures, communication delays, and others. The problem lies in that is that the properties of dependability and self-adaptivity are to an extent contradictory, as the easier it is for a system to adapt itself at runtime, the less predictable, thus less dependable it is. In response to this challenge, the report elaborates on *IRM for Self-Adaptivity* (IRM-SA), an extension of IRM to accommodate alternative system designs by allowing for alternative invariant decompositions. IRM-SA was originally proposed in [BGH+14b]. Dependability in IRM-SA is provided by tracing system configurations to high-level requirements through the decomposition links of the IRM-SA model (graph). Self-adaptivity is provided by switching between available configurations at runtime, corresponding to alternative decompositions in the IRM-SA model.

The report details a number of aspects of IRM-SA. First, it describes the IRM-SA modeling process (a preliminary description appeared in the paper of Section 4.5). Second, it describes how the idea of quantifying the inaccuracy of belief (originally proposed in [AABG+14b]) is integrated into the IRM-SA model in order to add to the dependability of siCPS by self-adapting in anticipation of critical situations. Third, the report describes a technical approach, based on monitoring of assumptions and SAT solving, of using the IRM-SA model at runtime as a self-adaptation mechanism. Fourth, it explains why self-adaptation in siCPS should be performed in a decentralized way.

On the evaluation side, the report details on how the IRM-SA self-adaptation mechanism is embedded in jDEECo, and how the experiments conducted with jDEECo provide evidence that IRM-SA can cope with operational uncertainty in decentralized deployments. It also describes an empirical study, which took the form of a controlled experiment, whose goal was to evaluate the effectiveness of the IRM-SA process.

Comments on Authorship

My personal contribution lies in elaborating the main idea, which involved extending IRM with alternative decompositions and using them at runtime to support runtime reconfiguration, and coming up with running example of the fire fighters collaboration system. I was responsible for documenting the IRM design process, performing the experimental evaluation (both the implementation of the IRM-jDEECo plugin and the empirical study), and positioning IRM-SA against the related work. Finally, under helpful guidance and supervision of the other authors, I authored a majority of the text.

Self-Adaptation in Cyber-Physical Systems: from System Goals to Architecture Configurations

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

Tomas Bures[1,2]
bures@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

Noël Plouzeau[3]
noel.plouzeau@irisa.fr

[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]IRISA
University of Rennes 1
Rennes, France

ABSTRACT

Design of self-adaptive Cyber-Physical Systems (CPS) operating in dynamic environments is a significant challenge when a sufficient level of dependability is required. This stems partly from the fact that the concerns of self-adaptivity and dependability are to an extent contradictory. In this paper, we introduce IRM-SA (Invariant Refinement Method for Self Adaptation) – a design method and associated formally grounded model targeting CPS – that addresses self-adaptivity and supports dependability by providing traceability between system requirements, distinct situations in the environment, and predefined configurations of system architecture. Additionally, IRM-SA allows for self-adaptation at runtime and copes with operational uncertainty, such as temporary disconnections and network delays. As a proof of concept, it was implemented in DEECo, a component framework that is based on dynamic ensembles of components. Furthermore, its feasibility was evaluated in experimental settings assuming decentralized system operation.

Keywords

Cyber-physical systems; Self-adaptivity; Dependability; System design; Component architectures

1. INTRODUCTION

Cyber-physical systems (CPS) are characterized by a network of distributed interacting elements which respond, by sensing and actuating, to activities in the physical world (their environments). Examples of CPS are numerous: systems for intelligent car navigation, smart electric grids, emergency coordination systems, to name just a few.

Design of such systems is a challenging task, as one has to deal with the different, and to an extent contradictory, concerns of dependability and self-adaptivity. Since they often host safety-critical applications, they need to be dependable (safe and predictable in the first place), even when being highly dynamic. Since CPS operate in adversarial environments (parts of the ever-changing physical world), they need be self-adaptive [44]. An additional issue is the inherent operational uncertainty related to their infrastructure. Indeed, modern CPS need to remain operable even in adversarial conditions, such as network unavailability, hardware failures, resource scarcity, etc.

Achieving synergy of dependability and self-adaptivity in presence of operational uncertainty is hard. Existing approaches typically succeed in addressing just one aspect of the problem. For example, agent-oriented methodologies address conceptual autonomy [18, 46]; component-based mode-switching methods bring dependability assurances with limited self-adaptivity [23, 30]. Operational uncertainty is typically viewed as a separate problem [15]. What is missing is a design method and associated model that would specifically target the development of dependable and self-adaptive CPS while addressing operational uncertainty.

Self-adaptive CPS need to be able to adapt to distinct runtime *situations* in the environment (i.e., its states as observed by the system). This takes the form of switching between distinct architecture *configurations* of the system (*products* in SPLs [2]). Being reflected in system requirements, these configurations and the associated situations can be systematically identified and addressed via requirement analysis and elaboration, similar to identification of adaptation scenarios in a Dynamically Adaptive System (DAS) [26].

A problem is that an exhaustive enumeration of configurations and situations at design time is not a viable solution in the domain of CPS, where unanticipated situations can appear in the environment (external uncertainty in [20]). Moreover, another challenge is that self-adaptive CPS need to base their adaptation actions not only on the current situation, but also on how well the system can currently operate, e.g., whether certain components can properly communicate (issue similar to agent capabilities [58]).

In this paper we tackle these challenges by proposing an extension to IRM [37] – a design method and associated formally grounded model targeting CPS requirements and architecture design. This extension (*IRM for Self-Adaptation – IRM-SA*) supports self-adaptivity and, at the same time, accounts for dependability. In particular, dependability is provided in the form of (i) traceability between system requirements and configurations, and (ii) mechanisms to deal with the operational uncertainty. Self-adaptivity is provided in the form of switching between configurations at runtime (*architecture adaptation*) to address specific situations.

To evaluate the feasibility of IRM-SA we have applied it on a firefighter coordination case study – *Firefighter Tactical Decision System* (FTDS) – developed within the project DAUM[1]. As proof of the concept, we implemented self-adaptation based on IRM-SA by extending DEECo [11] – a component model facilitating open-ended and highly dynamic CPS architectures. This allowed us to

[1] http://daum.gforge.inria.fr/

evaluate the capability of IRM-SA to cope with adversarial situations. We also evaluated the design process of IRM-SA via a controlled experiment.

In summary, key contributions of this paper include:

- The description of a design method and associated model that allows modeling design alternatives in the architecture of a CPS pertaining to distinct situations via systematic elaboration of system requirements;
- An automated self-adaptation method that selects the appropriate architecture configuration based on the modeled design alternatives and the perceived situation;
- An evaluation of how the proposed self-adaptation method deals with operational uncertainty in fully decentralized settings;
- A discussion of strategies to deal with unanticipated situations at design time and runtime.

The paper is structured as follows. Section 2 describes the running example, while Section 3 presents the background on which IRM-SA is based. Then, Section 4 overviews the core ideas of IRM-SA. Section 5 elaborates on the modeling of different design alternatives in IRM-SA by extending IRM, while Section 6 focuses on the selection of applicable architecture configurations at runtime. Section 7 describes an evaluation covering both (i) a realization of IRM-SA in the DEECo component system and experiments in decentralized simulation settings, and (ii) an empirical study of IRM-SA effectiveness via a controlled experiment. Section 8 is focused on the intricacies of self-adaptation in decentralized settings and coping with unanticipated situations. Section 9 discusses the related work, while Section 10 concludes the paper.

2. RUNNING EXAMPLE

In this paper, we use as running example a simple scenario from the FTDS case study, which was developed in cooperation with professional firefighters. In the scenario, the firefighters belonging to a tactical group communicate with their group leader. The leader aggregates the information about each group member's condition and his/her environment (parameters considered are firefighter *acceleration*, external *temperature*, *position* and *oxygen level*). This is done with the intention that the leader can infer whether any group member is in danger so that specific actions are to be taken to avoid casualties.

On the technical side, firefighters in the field communicate via low-power nodes integrated into their personal protective equipment. Each of these nodes is configured at runtime depending on the task assigned to its bearer. For example, a hazardous situation might need closer monitoring of a certain parameter (e.g., temperature). The group leaders are equipped with tablets; the software running on these tablets provides a model of the current situation (e.g., on a map) based on the data aggregated from the low-power nodes.

The main challenge of the case study is how to ensure that individual firefighters (nodes) retain their (a) autonomy so that they can operate in any situation, even entirely detached from the network and (b) autonomicity so that they can operate optimally without supervision, while still satisfying certain system-level constraints and goals. Examples of challenging scenarios include (i) loss of communication between a leader and members due to location constraints, (ii) malfunctioning of sensors due to extreme conditions or battery drainage, and (iii) data inaccuracy and obsoleteness due to intermittent connections. In all these cases, firefighters have to adjust their behavior according to the latest information available. Such adjustments range from simple adaptation actions (e.g., increasing the sensing rate in face of a danger) to complex cooperative actions (e.g., relying on the nearby nodes for strategic actions when communication with the group leader is lost).

3. BACKGROUND

Invariant Refinement Method (IRM) [37] is a goal-oriented design method targeting the domain of CPS. IRM builds on the idea of iterative refinement of system objectives yielding low-level obligations which can be operationalized by system components. Contrary to common goal-oriented modeling approaches (e.g., KAOS [40], Tropos/i* [8]), which focus entirely on the problem space and on stakeholders' intentions, IRM focuses on system components and their contribution and coordination in achieving system-level objectives. IRM also incorporates the notion of feedback loops present in autonomic and control systems, i.e. all "goals" in IRM are to be constantly maintained, not achieved just once. A key advantage of IRM is that it allows capturing the compliance of design decisions with the overall system goals and requirements; this allows for design validation and verification.

The main idea of IRM is to capture high-level system goals and requirements in terms of *invariants* and, by their systematic refinement, to identify system *components* and their desired interaction. In principle, invariants describe the *operational normalcy* of the system-to-be, i.e., the desired state of the system-to-be at every time instant. For example, the main goal of our running example is expressed by INV-1: "GL keeps track of the condition of his/her group's members" (Figure 1).

IRM invariants are agnostic on the language used for their specification. In this paper, plain English is used for simplicity's sake; passive voice has been chosen in order to adopt a more descriptive than prescriptive style. Other possible choices include adopting a style based on SHALL statements commonly used in requirements specifications, or a complete textual requirements specification language, such as RELAX [63].

In general, invariants are to be maintained by system components and their cooperation. At the design stage, a component is a participant/actor of the system-to-be, comprising internal state. Contrary to common goal-oriented approaches (e.g., [41], [8]), only software-controlled actors are considered. The two components identified in the running example are Firefighter and Officer.

As a special type of invariant, an *assumption* describes a condition expected to hold about the environment; an assumption is not expected to be maintained by the system-to-be. In the example, INV-8 in Figure 1 expresses what the designer assumes about the monitoring equipment (e.g., GPS).

As a design decision, the identified top-level invariants are decomposed via so-called *AND-decomposition* into conjunctions of more concrete sub-invariants represented by a decomposition model – *IRM model*. Formally, the IRM model is a directed acyclic graph (DAG) with potentially multiple top-level invariants, expressing concerns that are orthogonal. The *AND-decomposition* is essentially a refinement, where the composition (i.e., conjunction) of the children implies the fact expressed by the parent (i.e., the fact expressed by the composition is in general a specialization, following the traditional interpretation of refinement). Formally, an AND-decomposition of a parent

invariant I_p into the sub-invariants $I_{s1}, \ldots, I_{sn}$ is a refinement, if it holds that:

1. $I_{s1} \wedge \ldots \wedge I_{sn} \Rightarrow I_p$ (entailment)
2. $I_{s1} \wedge \ldots \wedge I_{sn} \not\Rightarrow false$ (consistency)

For example, the top-level invariant in Figure 1 is refined to express the necessity to keep the list of sensor data updated on the Officer's side (INV-4) and the necessity to filter the data to identify group members that are in danger (INV-5).

Decomposition steps ultimately lead to a level of abstraction where leaf invariants represent detailed design of the system constituents. There are two types of leaf invariants: *process invariants* (labeled P, e.g., INV-5) and *exchange invariants* (labeled X, e.g., INV-7). A process invariant is to be maintained by a single component (at runtime, in particular by a cyclic process manipulating the component's state – Section 7.1.1). Conversely, exchange invariants are maintained by component interaction, typically taking the form of knowledge exchange within a group of components (Section 7.1.1). In this case, exchange invariants express the necessity to keep a component's belief over another component's internal state. Here, *belief* is defined as a snapshot of another component's internal state [37] – often the case in systems of autonomous agents [58]; inherently, a belief can get outdated and needs to be systematically updated by knowledge exchange in the timeframe decided at the design stage.

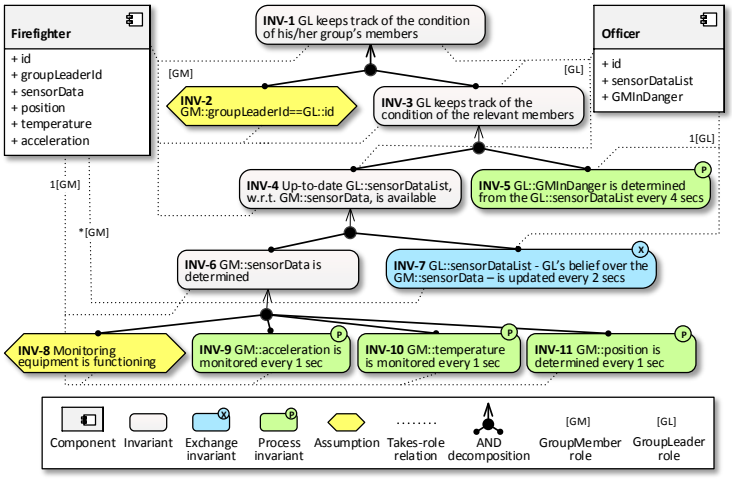


Figure 1: IRM decomposition of the running example.

4. IRM-SA – THE BIG PICTURE

To induce self-adaptivity by design so that the running system can adapt to situations, it is necessary to capture and exploit the architecture variability in situations that warrant self-adaptation. Specifically, the idea is to identify and map *applicable configurations* to situations by elaborating *design alternatives* (alternative realizations of system's requirements); then these applicable configurations can be employed for architecture adaptation at runtime.

Therefore, we extended the IRM design model and process to capture the design alternatives and applicable configurations along with their corresponding situations. For each situation there can be one or more applicable configurations. To deal with operational uncertainty, we also extend the model by reasoning on the inaccuracies of the belief.

At runtime, the actual architecture self-adaptation is performed via three recurrent steps: (i) determining the current situation, (ii) selecting one of the applicable configurations, and (iii) reconfiguring the architecture towards the selected configuration.

The challenge is that mapping configurations to situations typically involves elaborating a large number of design alternatives. This creates a scalability issue both at design-time and runtime, especially when the individual design alternatives have mutual dependencies or refer to different and possibly nested levels of abstraction. To address scalability, we employ (i) separation of concerns via decomposition at design time; (ii) a formal base of the IRM-SA design model and efficient reasoning based on SAT solving for the selection of applicable configurations at runtime (Section 6).

5. MODELING DESIGN ALTERNATIVES

5.1 Concepts addressing Self-Adaptivity

Although providing a suitable formal base for architecture design via elaboration of requirements, IRM in its pure form does not allow modeling of design alternatives.

Therefore, we extended the IRM design model with the concepts of alternative decomposition – *OR-decomposition* – and *situation*. The running example as modelled in IRM-SA is depicted in Figure 2.

Essentially, OR-decomposition denotes a variation point where each of the children represents a design alternative. Technically, OR-decomposition is a refinement, where each of the children individually implies (i.e., refines) the fact expressed by the parent. OR-decompositions can be nested, i.e., a design alternative can be further refined via another OR-decomposition. Formally, an OR-decomposition of a parent invariant I_p into the sub-invariants $I_{s1}, \ldots, I_{sn}$ is a refinement if it holds that:

1. $I_{s1} \vee \ldots \vee I_{sn} \Rightarrow I_p$ (alternative entailment)
2. $I_{s1} \vee \ldots \vee I_{sn} \not\Rightarrow false$ (alternative consistency)

Each design alternative addresses a specific situation which is characterized via an assumption. Thus, each I_{si} contains at its top-most level a *characterizing assumption* as illustrated in Figure 2.

For example, consider the left-most part of the refinement of INV-3: "GL keeps track of the condition of the relevant members", which captures two design alternatives corresponding to the situations where either some firefighter in the group is in danger or none is. In the former case (left alternative), INV-7 is also included – expressing the necessity to inform the other firefighters in the group that a member is in danger. In this case, INV-6 and INV-8 are the characterizing assumptions in this OR-decomposition.

The situations addressed in an OR-decomposition may overlap, i.e. their charactering assumption can hold at the same time. This is the case of INV-13 and INV-18 capturing that both one Firefighter is in danger and a nearby colleague as well. Consequently, there are more than one applicable configurations and therefore a prioritization is needed (Section 6.2). As an aside,

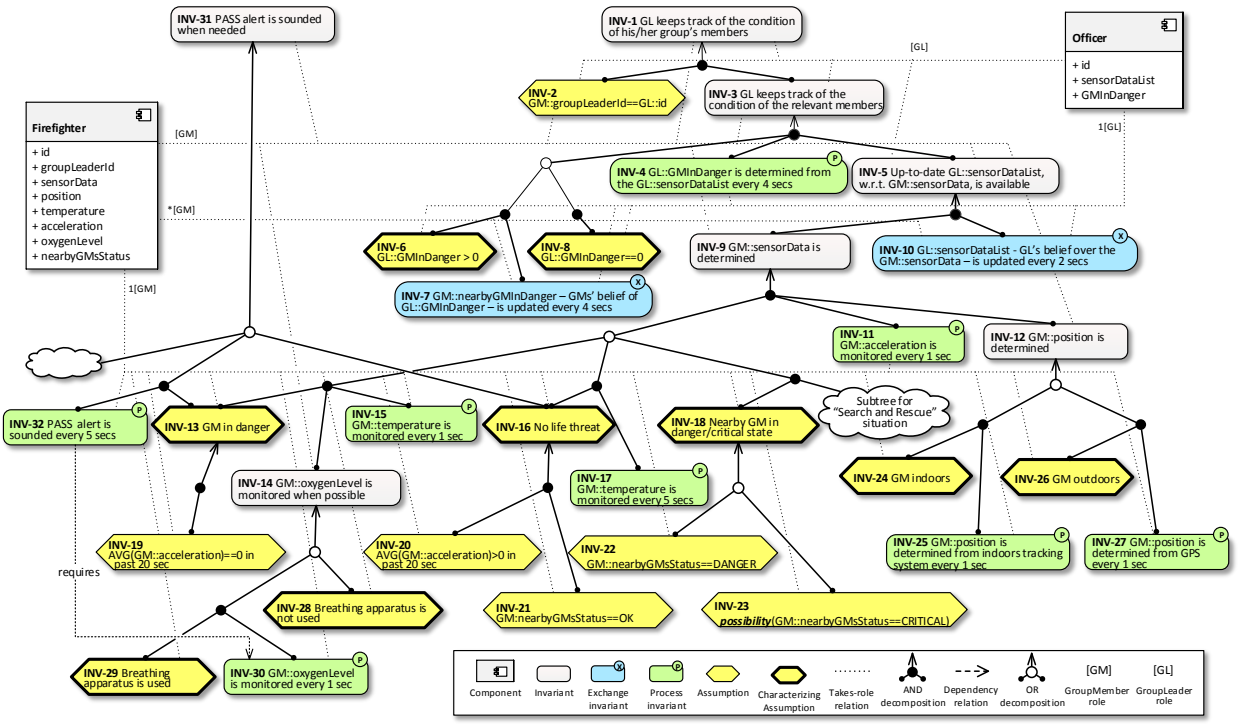


Figure 2: IRM-SA model of the running example.

allowing situations in an OR-decomposition to overlap also provides a built-in fault-tolerance mechanism (Section 8.1.1).

Technically, if a design alternative in an OR-decomposition is further refined in terms of an AND-decomposition (or vice-versa), we omit the invariant representing the alternative and connect the AND-decomposition directly to the OR-decomposition to improve readability (e.g., design alternatives of INV-12).

We distinguish two kinds of invariants: *computable* and *non-computable*. While a computable invariant can be programmatically evaluated at runtime, a non-computable invariant serves primarily for design and review-based validation purposes. Thus, the characterizing assumptions need to be either computable or decomposed into computable assumptions.

An example of a non-computable characterizing assumption is INV-16: "No life threat". It is AND-decomposed into the computable assumptions INV-20 and INV-21, representing two orthogonal concerns, which can be evaluated by monitoring the Firefighter's internal state.

Dependencies may also exist between invariants in design alternatives across different OR-decompositions (*cross-tree dependencies*), reflecting constraints of the physical world. These dependencies are captured in the IRM-SA model by directed links between invariants labeled with "requires", resp. "collides", which capture the constraint that the source invariant can appear in a configuration only with, resp. without, the target invariant. For example, in order for INV-32 to appear in a configuration, INV-30 has to be included as well, capturing the real-life constraint where the *Personal Alert Safety System* (PASS) is attached to the *self-contained breathing apparatus* (SCBA) of a firefighter; thus if the SCBA is not used, then the PASS cannot be used as well. The "collides" dependency is not illustrated in our running example.

5.2 Concepts Addressing Dependability

In IRM-SA, dependability is mainly pursued by tracing the low-level processes to high-level invariants. Moreover, to deal with the operational uncertainty in dynamic CPS, IRM-SA goes beyond the classical goal-modeling approaches and allows self-adaptation based not only on valuations of belief (snapshot of a remote component's internal data), but also on valuations of associated metadata (timestamp of belief, timestamp of sending/receiving the belief over the network, etc.). This functionality also adds to the dependability by self-adapting in anticipation of critical situations. Nevertheless, IRM-SA support for dependability does not cover other dependability aspects, such as privacy and security.

A key property here is that a belief is necessarily outdated, because of the distribution and periodic nature of real-time sensing in CPS. For example, the position of a Firefighter as perceived by his/her Officer would necessarily deviate from the actual position of the Firefighter if he/she were on the move. Instead of reasoning directly on the degree of belief outdatedness (on the time domain), we rely on models that predict the evolution of the real state (e.g., state-space models if this evolution is governed by a physical process), translate the outdatedness from the time domain to the data domain of the belief (e.g. position inaccuracy in meters) and reason on the degree of *belief inaccuracy*. For this, we build on our previous work in quantifying the degree of belief inaccuracy in dynamic CPS architectures [1].

For illustration, consider an assumption "*inaccuracy*(GM::position) < 20 m", which models the situation where the difference of the measured and actual positions is less than 20 meters. In this case, belief inaccuracy is both (i) inherent to the sensing method (more GPS satellites visible determine more

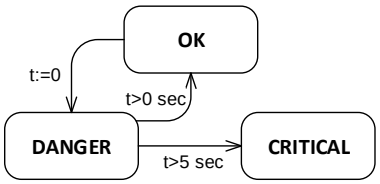


Figure 3: Timed automaton capturing the transitions in the valuation of the nearbyGMsStatus field.

accurate position), and (ii) related to the network latencies when disseminating the position data (more outdated data yield more inaccurate positions – since firefighters are typically on the move, their position data are subject to outdating). As a result, an Officer has to reason on the cumulative inaccuracy of the position of his/her Firefighter.

When the domain of the belief field is discrete instead of continuous, we rely on models that capture the evolution of discrete values in time, such as timed automata. For illustration, consider assumption INV-23: "*possibility*(GM::nearbyGMsStatus == CRITICAL)", which models the situation where the nearbyGMsStatus enumeration field with values OK, DANGER, and CRITICAL is possible to evaluate to CRITICAL. This presumes that the designer relies on a simple timed automaton such as the one depicted in Figure 3,which encodes the domain knowledge that a firefighter gets into a critical situation (and needs rescuing) at least 5 seconds after he/she gets in danger.

All in all, the invariants that are formulated with *inaccuracy* and *possibility* provide a fail-safe mechanism for adversarial situations, when the belief of a component gets so inaccurate that special adaptation actions have to be triggered. This adds to the overall dependability of the self-adaptive CPS.

5.3 The Modeling Process

As is usually the case with software engineering processes, IRM-SA modeling process is a mixed top-down and bottom-up process. As input, the process requires a set of use cases/user stories covering both the main success scenarios and the associated extensions. The main steps of the process are illustrated in Figure 4. After the identification of the main situations, goals and components, the architect starts to specify the knowledge of each component together with its takes-role relations (step 4), while, in parallel, he/she starts refining the invariants (step 5). These two steps require potentially several iterations to complete. In step 6, the architect composes the dangling invariant trees that may have resulted from the previous steps, i.e., the trees the roots of which are *not* top-level invariants. Contrary to the previous steps, this is a bottom-up design activity. In the final steps, as an optimization, the subtrees produced in the previous steps that are identical are merged together, and requires/collides dependencies are added. The result is a DAG – this optimization was applied also in Figure 2.

The workings of a single refinement are depicted in Figure 5. Based on the whether the invariant under question is to be satisfied in a different way in different situations (e.g. "position reading" will be satisfied by using the GPS sensor when outdoors and by using the indoor positioning system when indoors), the architect chooses to refine the invariant by OR- or AND-decomposition. Obviously, in the former case, the refinement involves specifying the characterizing assumption for each design alternative (situation). Note that, if the characterizing assumption

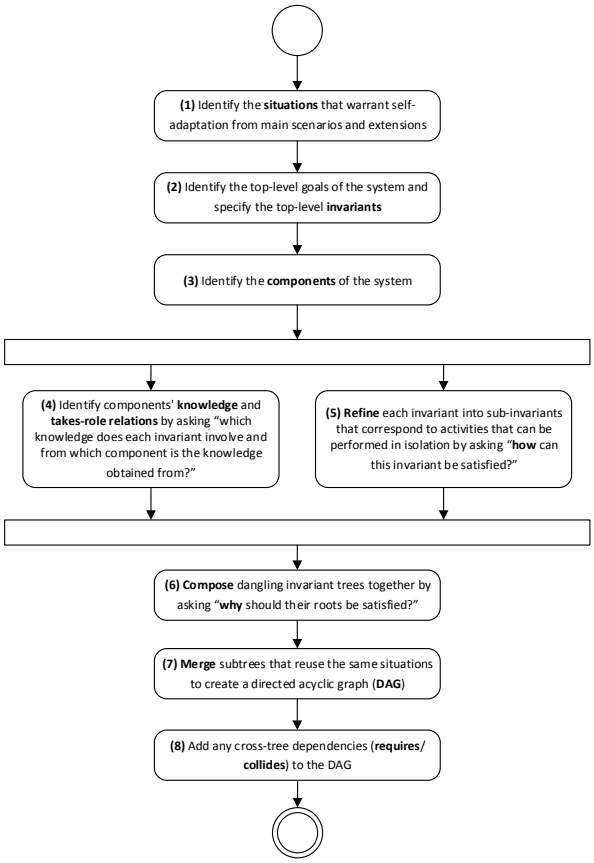


Figure 4: Steps in the IRM-SA modeling process.

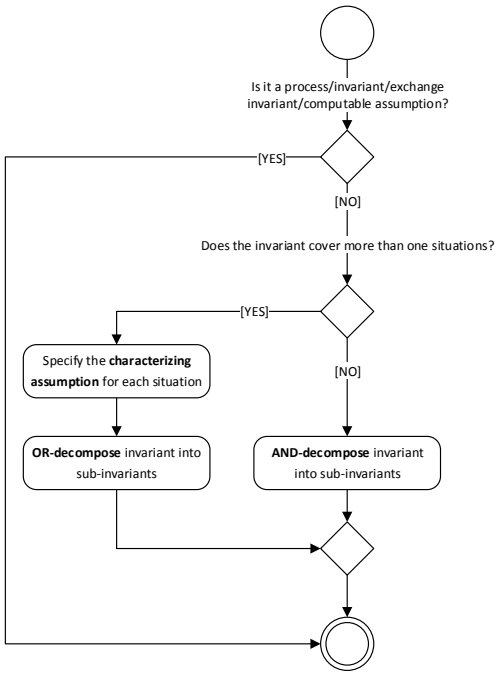


Figure 5: Steps in a single invariant refinement.

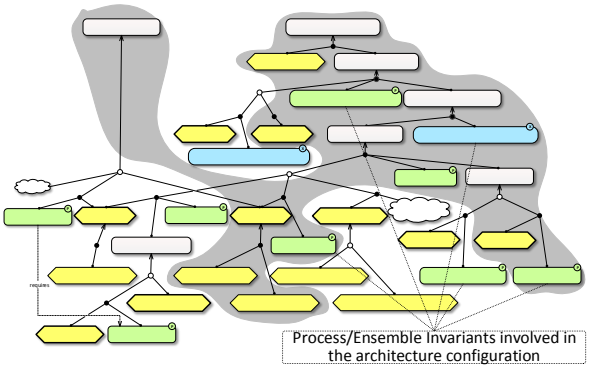


Figure 6: An architecture configuration of the running example.

is not computable, it will get refined in a next step as any other invariant in such a case.

The process of systematic identification of all the possible variation points and modeling of the corresponding design alternatives and situations is closely related to the identification of adaptation scenarios in a Dynamically Adaptive System (DAS) [26]. Here, one can leverage existing approaches in requirements engineering ranging from documentation of main use-case scenarios and extensions to obstacle/threat analysis on goal models [42]. Performance and resource optimization concerns can also guide the identification of variation points and corresponding design alternatives.

For example, the rationale behind the OR-decomposition of the left-most part of the AND-decomposition of INV-9 is resource optimization: under normal conditions the accuracy of external temperature monitoring can be traded off for battery consumption of the low-power node; this, however, does not hold in danger (e.g., a firefighter is not moving, INV-19), when higher accuracy of external temperature monitoring is needed.

On the contrary, the OR-decomposition of INV-12 has its rationale in a functional constraint: since GPS is usually not available within a building, a firefighter's position has to be monitored differently in such a case, e.g., through an indoors tracking system [13]. This is an example of a technology-driven process of identification of design alternatives, where the underlying infrastructure significantly influences the possible range of adaptation scenarios [26]. For example, it would not make sense to differentiate between the situations of being indoors and outdoors, if there were no way to mitigate the "GPS lost signal" problem using the available infrastructure.

This highlights an important point: IRM-SA allows for modeling the environment via assumptions, but, at the same time, guides the designer into specifying only the pertinent features of the environment, avoiding over-specification.

For a complete example of this modeling process, we refer the reader to the online IRM-SA User Guide[2]. To support the modeling process, we have also developed a prototype of a GMF-based IRM-SA design tool [33].

[2] http://www.ascens-ist.eu/irm

6. SELECTING ARCHITECTURE CONFIGURATIONS BY SAT SOLVING

As outlined in Section 4, given an IRM-SA model, the selection of a configuration for a situation can be advantageously done by directly reasoning on the IRM-SA model at runtime. In this section we describe how we encode the problem of selecting an applicable configuration into a Boolean satisfiability (SAT) problem (6.1), our prioritizing strategy with multiple applicable configurations (6.2), and how we bind variables in the SAT instance based on monitoring (6.3).

To simplify the explanation, we use the term "clause" in this section even for formulas which are not necessarily valid clauses in the sense of CNF (Conjunctive Normal Form – the default input format for SAT), but rely on the well-known fact that every propositional formula can be converted to an equisatisfiable CNF formula in polynomial time.

6.1 Applicable Configurations

Formally, the problem of selecting an applicable configuration is the problem of constructing a set C of selected invariants from an IRM-SA model such that the following rules are satisfied: (i) all the top-level invariants are in C; (ii) if an invariant I_p is decomposed by an AND-decomposition to $I_1, \dots, I_m$, then $I_p \in C$

1. // 1. configuration constraints based of the IRM model
2. // top level decomposition in Figure 2
3. $s_{11\_1} \wedge s_{27} \wedge s_{28} \Leftrightarrow s_{11}$ // s_{11_1} represents the anonymous invariant in the AND decomposition of INV-9
4. $s_{33\_1} \vee s_{33\_2} \vee s'_{20} \Leftrightarrow s_{33}$ // s'_{20} is a copy of s_{20}
5.
6. // decomposition level 1 in Figure 2
7. $s_{11\_1\_1} \vee s_{11\_1\_2} \vee s_{11\_1\_3} \Leftrightarrow s_{11\_1}$
8. $s_{28\_1} \vee s_{28\_2} \Leftrightarrow s_{28}$
9. $s_{34} \wedge s'_{13} \Leftrightarrow s_{33\_2}$ // s'_{13} is a copy of s_{13}
10.
11. // decomposition level 2 in Figure 2
12. $s_{13} \wedge s_{14} \wedge s_{15} \Leftrightarrow s_{11\_1\_1}$
13. $s_{20} \wedge s_{21} \Leftrightarrow s_{11\_1\_2}$
14. $s_{24} \wedge \dots \Leftrightarrow s_{11\_1\_3}$
15. $s_{30} \wedge s_{29} \Leftrightarrow s_{28\_1}$
16. $s_{32} \wedge s_{31} \Leftrightarrow s_{28\_2}$
17.
18. // decomposition level 3 in Figure 2
19. $s_{16} \Leftrightarrow s_{13}$
20. $s'_{16} \Leftrightarrow s'_{13}$
21. $s_{14\_1} \vee s_{19} \Leftrightarrow s_{14}$
22. $s_{22} \wedge s_{23} \Leftrightarrow s_{20}$
23. … // similar for s'_{20}, s_{24}
24.
25. // decomposition level 4 in Figure 2
26. $s_{17} \wedge s_{18} \Leftrightarrow s_{14\_1}$
27.
28. // 2. only applicable invariants may be selected into a configuration
29. $(s_{11} \Rightarrow a_{11}) \wedge \dots \wedge (s_{32} \Rightarrow a_{32}) \wedge \dots$
30.
31. // 3. determining acceptability according to monitoring
32. // (current configuration as shown in Figure 6)
33. // 3.1. active monitoring
34. $a_{11} = \cdots$ // true or false based on the monitoring of INV-9
35. … // repeat for $a_{16}, a'_{16}, a_{17}, a_{19}, a_{21}, a_{22}, a'_{22}, a_{23}, a'_{23}, a_{25}, a_{27}, a_{28}, a_{32}, a_{33}$
36.
37. // 3.2. predictive monitoring
38. $a_{15} = \cdots$
39. … // repeat for the rest

Figure 7: Encoding the IRM-SA model of running example into SAT.

iff all $I_1, \dots, I_m \in C$; (iii) if an invariant I_p is decomposed by an OR-decomposition to $I_1, \dots, I_m$, then $I_p \in C$ iff at least one of $I_1, \dots, I_m$ is in C; (iv) if an invariant I_p requires, resp. collides (with), I_q, then $I_p \in C$ iff $I_p \in C$, resp. $I_p \notin C$. The set C represents an applicable configuration. The rules above ensure that C is well-formed with respect to decomposition and cross-tree dependencies semantics. Figure 6 shows a sample applicable configuration (selected invariants are outlined in grey background).

Technically, for the sake of encoding configuration selection as a SAT problem, we first transform the IRM-SA model to a forest by duplicating invariants on shared paths. (This is possible because the IRM-SA model is a DAG.) Then we encode the configuration C we are looking for by introducing Boolean variables $s_1, \dots, s_n$, such that $s_i = true$ iff $I_i \in C$. To ensure C is well-formed, we introduce clauses over $s_1, \dots, s_n$ reflecting the rules (i)-(iv) above. For instance, the IRM-SA model from Figure 2 will be encoded as shown in Figure 7, lines 1-26.

To ensure that C is an applicable configuration w.r.t. a given situation, we introduce Boolean variables $a_1, \dots, a_n$ and add a clause $s_i \Rightarrow a_i$ for each $i \in \{1 \dots n\}$ (Figure 7, line 29). The value of a_i captures whether the invariant I_i is *acceptable*; i.e., $true$ indicates that it can be potentially included in C, $false$ indicates otherwise. The variables $a_1, \dots, a_n$ are bound to reflect the state of the system and environment (Figure 7, lines 31-39). This binding is described in Section 6.3.

In the resulting SAT instance, the variable s_t for each top-level invariant I_t is bound to true to enforce the selection of at least one applicable configuration. A satisfying valuation of such a SAT instance encodes one applicable configuration (or more than one in case of overlapping situations – see Section 6.2), while unsatisfiability of the instance indicates nonexistence of an applicable configuration in the current situation.

6.2 Prioritizing Applicable Configurations

Since the situations in an OR-decomposition do not need to be exclusive but can overlap, SAT solving could yield more than one applicable configurations. In this case, we assume a post-processing process that takes as input the IRM-SA model with the applicable configurations and outputs the selected configuration based on analysis of preferences between design alternatives. For this purpose, one can use strategies that range from simple total preorder of alternatives in each decomposition to well-established soft-goal-based techniques for reasoning on goal-models [16]. In the rest of the section, we detail on the prioritization strategy used in our experiments, which we view as just one of the many possible.

For experimental evaluation (Section 7.1.2), we have used a simple prioritization strategy based on total preorder of design alternatives in each OR-decomposition. Here, for simplicity, a total preorder - numerical ranking is considered (1 corresponds to the most preferred design alternative, 2 to the second most preferred, etc.). The main idea of the strategy is that the preferences loose significance by an order of magnitude from top to bottom, i.e., preferences of design alternatives that are lower in an IRM-SA tree cannot impact the selection of a design alternative that is above them on a path from the top-level invariant.

More precisely, given an IRM-SA tree, every sub-invariant I_i of an OR-decomposition is associated with its OR-level number d_i, which expresses that I_i is a design alternative of a d_i-th OR-decomposition on a path from the top-level invariant (level 1) to a leaf. For each OR-level, there is its cost base b_j defined in the following way: (a) the lowest OR-level has cost base equal to 1, (b) the j-th OR-level has its cost base $b_j = b_{j+1} * (n_{j+1} + 1)$, where n_{j+1} denotes the number of all design alternatives at the level $j + 1$ (i.e., considering all OR-decomposition at this level). For example, the 2nd OR-level in the running example has $b_2 = b_3 * (n_3 + 1) = 1 * (4 + 1) = 5$, since the 3rd OR-level (lowest) has in total 4 design alternatives (2 from the OR-decomposition of INV-14 and 2 from that of INV-18).

Having calculated the base for each OR-level, the cost of a child invariant I_i of a d_i-th OR-decomposition with a cost b_{d_i} is defined as $rank * b_{d_i}$, where $rank$ denotes the rank of the design alternative that the invariant I_i corresponds to. Finally, a simple graph traversal algorithm is the employed to calculate the cost of each applicable configuration as the sum of the cost of the selected invariants in the applicable configuration. The applicable configuration with the smallest cost is the preferred one – becomes the *current configuration*.

6.3 Determining Acceptability

Determining acceptability of an invariant I_i (i.e., determining the valuation of a_i) is an essential step. In principle, a valuation of a_i reflects whether I_i is applicable w.r.t. the current state of the system and the current situation. Essentially, $a_i = false$ implies that I_i cannot infer an applicable configuration.

We determine the valuation of a_i in one of the following ways (alternatively):

(1) *Active monitoring*. If I_i belongs to the current configuration and is computable, we determine a_i by evaluating I_i w.r.t. the current knowledge of the components taking a role in I_i.

(2) *Predictive monitoring*. If I_i does not belong to the current configuration and is computable, it is assessed whether I_i would be satisfied in another configuration if chosen.

In principle, if I_i is not computable, its acceptability can be inferred from the computable sub-invariants.

For predictive monitoring, two evaluation approaches are employed: (a) The invariant to be monitored is extended by a *monitor predicate* (which is translated into a monitor – Section 7.1.1) that assesses whether the invariant would be satisfied if selected, and (b) the history of the invariant evaluation is observed in order to prevent oscillations in current configuration settings by remembering that active monitoring found an invariant not acceptable in the past.

Certainly, (a) provides higher accuracy and thus is the preferred option. It is especially useful for process invariants, where the monitor predicate may assess not only the validity of process invariant (e.g. by looking at knowledge valuations of the component that take a role in it), but also whether the underlying process would be able to perform its computation at all. This can be illustrated on the process invariant INV-27, where the process maintaining it can successfully complete (and thus satisfy the invariant) only if GPS is operational and at least three GPS satellites are visible.

6.4 Decentralized Reasoning

Though the selection of an applicable configuration is relatively straightforward from the abstract perspective of SAT formulation, its actual decentralized implementation is more complex. This is because of the inherent distribution and dynamicity of CPS, which

```
1.  role PositionSensor:
2.    missionID, position
3.
4.  role PositionAggregator:
5.    missionID, positions
6.
7.  component Firefighter42 features PositionSensor, ...:
8.    knowledge:
9.      ID = 42, missionID = 2, position = {51.083582, 17.481073}, ...
10.   process measurePositionGPS (out position):
11.     position ← Sensor.read()
12.   scheduling: periodic( 500ms )
13.   ... /* other process definitions */
14.
15. component Officer13 features PositionAggregator, ...:
16.   knowledge:
17.     ID = 13, missionID = 2, position = {51.078122, 17.485260},
18.       firefightersNearBy = {42, ...},  positions = {{42, {51.083582,
19.         17.481073}},...}
20.   process findFirefightersNearBy(in positions, in position, out
21.     firefightersNearBy):
22.       firefightersNearBy← checkDistance(position, positions)
23.     scheduling: periodic( 1000ms )
24.   ... /* other process definitions */
25. ... /* other component definitions */
26.
27. ensemble PositionUpdate:
28.   coordinator: PositionAggregator
29.   member: PositionSensor
30.   membership:
31.     member.missionID == coordinator.missionID
32.   knowledge exchange:
33.     coordinator.positions ← { (m.ID, m.position) | m ∈ members }
34.   scheduling: periodic( 1000ms )
35. ... /* other ensemble definitions */
```

Figure 8: Example of possible DEECo components and ensembles in the running example.

in particular implies that: (1) communication is inherently unreliable; (2) all data perceived by components are inevitably to an extent outdated; (3) the whole CPS has to observe and control the environment in real-time, no matter whether it was able to coordinate the selection and execution of the applicable configuration or not.

Such a context makes it rather impossible to employ any solution which relies on a strict synchronization of components' knowledge – such as a selected arbiter would collect all the necessary knowledge from components, solve the SAT problem, and reliably communicate the result back to the components (the classical centralized approach). Interestingly enough, even the decentralized approaches requiring distributed consensus fail in this case because of unbounded message delays.

Instead of trying to reach consensus among components, it is sufficient that each node has up-to-date enough data and solves independently the same SAT problem. Since SAT solving is deterministic, all nodes are supposed to reach the same solution (assuming they employ the same prioritization strategy).

The problem is that this would require zero communication delay though – something that is certainly not true in a real CPS infrastructure. Therefore we exploit the fact that temporary de-synchronization of components' knowledge is not harmful in major cases, but only reduces the overall performance. We provide a discussion on the factors that characterize the impact on performance in Section 8.2.

7. EXPERIMENTAL EVALUATION

The evaluation of IRM-SA has been two-fold – experiments of applicability of IRM-SA self-adaptation in practical settings

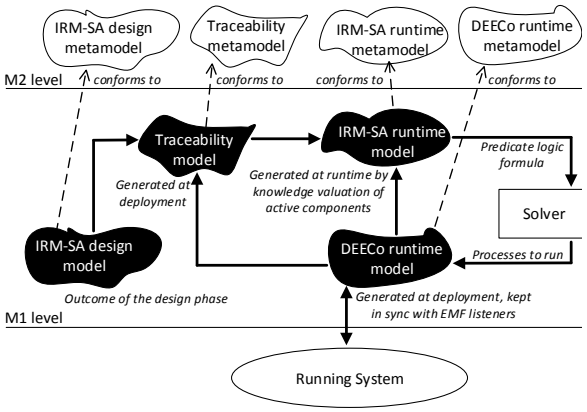


Figure 9: Models and their meta-models employed for self-adaptation in jDEECo.

(Section 8.1), and small-scale empirical study on the feasibility and effectiveness of the IRM-SA modeling process (Section 8.2).

7.1 Self-adaptation in jDEECo

7.1.1 Realization

We implemented the self-adaptation method of IRM-SA as a plugin (publicly available [33]) into the jDEECo framework [34]. This framework is a realization of the DEECo component model [11, 38] . For developing components and ensembles, jDEECo provides an internal Java DSL and allows their distributed execution.

DEECo Component Model. Dependable Emergent Ensemble of Components (DEECo) is component model (including specification of deployment and runtime computation semantics) tailored for building CPS with a high degree of dynamicity in their operation. In DEECo, *components* are autonomous units of deployment and computation. Each of them comprises *knowledge* and *processes*. Knowledge is a hierarchical data structure representing the internal state of the component. A process operates upon the knowledge and features cyclic execution based on the concept of feedback loop [50], being thus similar to a process in real-time systems. As an example, consider the two DEECo components in Figure 8, lines 7-13 and 15-23. They also illustrate that separation of concerns is brought to such extent that individual components do not explicitly communicate with each other. Instead, interaction among components is determined by their composition into *ensembles* – groups of components cooperating to achieve a particular goal [19, 31] (e.g., PositionUpdate ensemble in Figure 8, lines 27-34). Ensembles are dynamically established/disbanded based on the state of components and external situation (e.g., when a group of firefighters are physically close together, they form an ensemble). At runtime, *knowledge exchange* is performed between the components within an ensemble (lines 32-33) – essentially updating their beliefs (Section 3).

jDEECo Runtime. Each node in a jDEECo application contains a jDEECo runtime, which in turn contains one or more local components – serving as a container (Figure 10). The runtime is responsible for periodical scheduling of component processes and knowledge exchange functions (lines 12, 23, 34). It also possesses reflective capabilities in the form of a *DEECo runtime model* that provides runtime support for dynamic reconfigurations (e.g., starting and stopping of process scheduling). Each runtime manages the knowledge of both *local components*, i.e.,

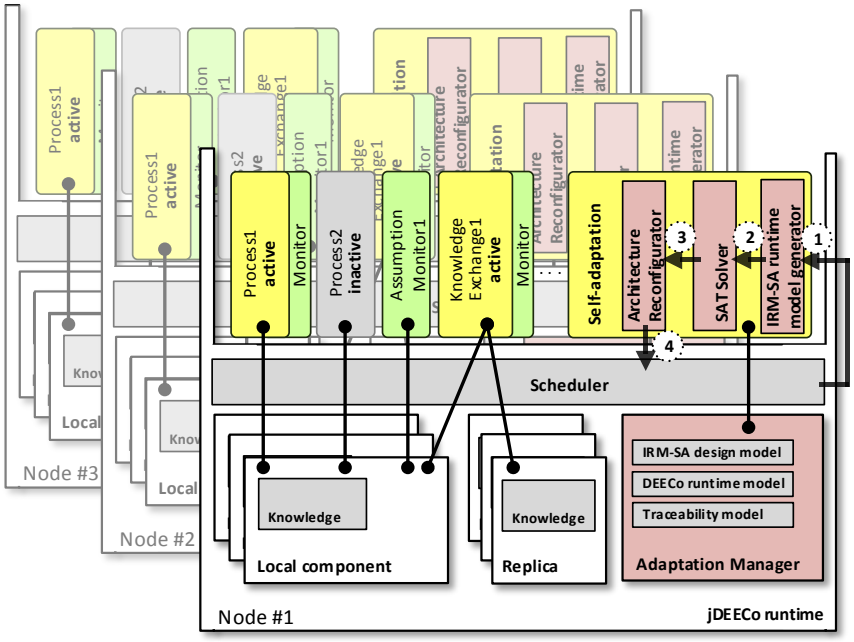


Figure 10: Steps in jDEECo self-adaptation: (1) Aggregate monitoring results from local component and replicas and create IRM-SA runtime model; (2) Translate into SAT formula, bind monitoring variables; (3) Translate SAT solver output to current configuration; (4) Activate/ deactivate component processes and knowledge exchange processes according to current configuration. At each node, self-adaptation is a periodically-invoked process of the "Adaptation Manager" system component (which is deployed on each node along with the application components).

components deployed on the same node, and *replicas*, i.e., copies of knowledge of the components that are deployed on different nodes but interact with the local components via ensembles.

Self-Adaptation in jDEECo. For integration of the self-adaptation method of IRM-SA *(jDEECo self-adaptation*) with jDEECo, a models-at-runtime approach [47] is employed, leveraging on EMF-based models (Figure 9). In particular, a *Traceability model* is created at deployment, providing the association of entities of the DEECo runtime model (i.e., components, component processes, and ensembles) with the corresponding constructs of the *IRM-SA design model* (i.e., components and invariants). This allows traceability between entities of requirements, design, and implementation – a feature essential for self-adaptation. For example, the process measurePositionGPS in Figure 11 is traced back to INV-27 (line 7), while the Firefighter component is traced back to its IRM-SA counterpart (line 2). Based on the Traceability model and the DEECo runtime model, an *IRM-SA runtime model* is generated by "instantiating" the IRM-SA design components with local components and replicas. Once the IRM-SA runtime model gets used for selecting an architecture configuration, the selected configuration is imposed to the DEECo runtime model as the current one.

A central role in performing jDEECo self-adaptation is played by a specialized jDEECo component Adaptation Manager (AM). Its functionality comprises the following steps (Figure 10): (1) Aggregation of monitoring results from local components and replicas and creation of IRM-SA runtime model. (2) Invocation of the SAT solver (Sections 6.1-6.3). (3) Translation of the SAT solver output into an applicable configuration (including prioritization). (4) Triggering the actual architecture adaptation – applying the current configuration. As an aside, internally, AM employs the SAT4J solver [43], mainly due to its seamless integration with Java.

The essence of step (4) lies in instrumenting the scheduler of the jDEECo runtime. Specifically, for every process, resp. exchange invariant in the current configuration, AM starts/resumes the scheduling of the associated component process, resp. knowledge exchange function. The other processes and knowledge exchange

```
1.  @Component
2.  @IRMComponent("Firefighter")
3.  public class Firefighter {
4.    public Position position;
5.    ...
6.
7.    @Invariant("27")
8.    @PeriodicScheduling(period=500)
9.    public static void monitorPositionGPS(
10.     @Out("position") Position position
11.   ) {
12.     // read current position from the GPS device
13.   }
14.   ...
15.
16.   @InvariantMonitor("27")
17.   public static boolean monitorPositionGPSMonitor(
18.     @In("position") Position position
19.   ) {
20.     // check health of GPS device
21.     // check if at least 3 satellites are visible
22.   }
23.   ...
24. }
```

Figure 11: Firefighter component definition and process-monitor definition in the internal Java DSL of jDEECo.

functions are not scheduled any more.

Monitoring. AM can handle both active and predictive monitoring techniques (Section 6.3). In the experiments described in Section 7.1.2, predictive monitoring was used for both component processes and knowledge exchange functions (based on observing the history of invariants evaluation), while for assumptions, only active monitoring was employed.

Technically, monitors are realized as Boolean methods associated with invariants in the Traceability model. For instance, a monitor for INV-27: "GM::position is determined from GPS" is illustrated in Figure 11, lines 16-22; it checks whether the corresponding process operates correctly by checking the health of the GPS device and the number of available satellites. The execution of monitors is driven directly by AM and is part of the first step of the jDEECo self-adaptation (Figure 10).

7.1.2 Coping with Operational Uncertainty

A key goal of the experiments was to support the claim "temporary de-synchronization of components' knowledge is not harmful in major cases, but only reduces the overall performance of the system compared to the ideal case of zero communication delay".

More specifically, the questions that we investigated by experiments were the following.

- Q1: *Do temporary network disconnections (and associated communication delays)* ***reduce*** *the overall performance of an application that employs jDEECo self-adaptation?*
- Q2: *Does using the IRM-SA method to cope with operational uncertainty* ***increase*** *the overall performance of an application that employs jDEECo self-adaptation?*

In the experiments we employed an extended version of the running example. This IRM-SA model consists of 4 components, 39 invariants, and 23 decompositions [33]. In the experiments, teams of firefighters consisting of three members and one leader were considered. In the simulated scenario, a firefighter A senses temperature higher than a pre-specified threshold (indication of being "in danger"); this information is propagated to the A's leader who in turn propagates the information that A is in danger to a firefighter B; then, B performs self-adaptation in the anticipation of the harmful situation of a having a group member in danger (*proactive self-adaptation*) and switches the mode to "Search and Rescue" (the situation captured by INV-18 in Figure 2). At the point when the leader determines that A is in danger, a temporary network disconnection occurs. The overall performance was measured by *reaction time* – the interval between the time that A sensed high temperature and the time that B switches to "Search and Rescue". Note that the overall performance corresponds to the cumulative utility function $u(a, \Delta t_{\max})$, as characterized in Section 8.2.

Simulation setup. The experiments were carried out using the jDEECo simulation engine [34]. Several simulation parameters (such as interleaving of scheduled processes) that were not relevant to our experiment goals were set to firm values. The simulation code, along with all the parameters and raw measurements, are available online [33].

To obtain a baseline, the case of no network disconnections was also measured. The result is depicted in dashed line in Figure 12.

To investigate Q1 and Q2, a number of network disconnections with preset lengths were considered; this was based on a prior

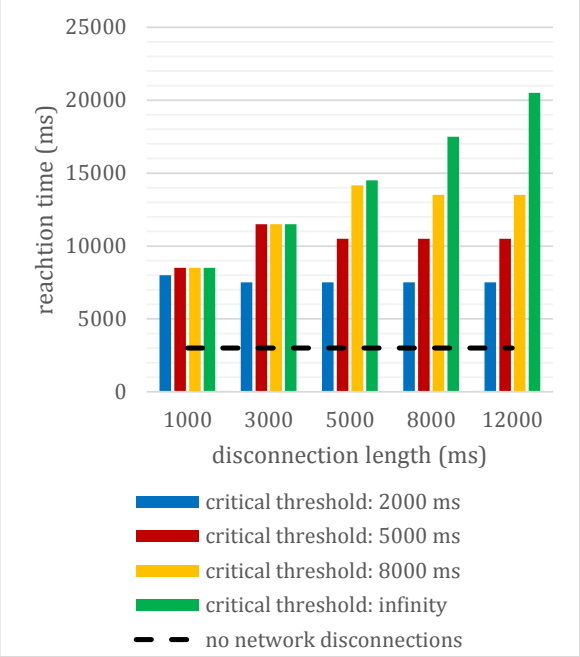


Figure 12: Reaction times for different network disconnection lengths and different critical thresholds. The results for each case have been averaged for different DEECo component knowledge publishing periods (400, 500 and 600 ms).

experience of working with deployment of DEECo on mobile ad-hoc networks [10].

To answer Q2, the timed automaton (Figure 3) associated with INV-23: "possibility(GM::nearbyGMsStatus == CRITICAL)" was modified: the transition from DANGER to CRITICAL was made parametric to experiment with different critical threshold values – critical threshold in the context of the experiments is the least time needed for a firefighter to get into a critical situation after he/she gets in danger (in Figure 3 the critical threshold is set to 5 sec). The reaction times for different critical thresholds and different disconnection lengths are in Figure 12.

To answer Q1 (as well as obtain the baseline), the critical threshold was set to infinity – effectively omitting INV-23 from the IRM-SA model – in order to measure the vanilla case where self-adaptation is based only on the values of data transmitted (belief) and not on other parameters such as belief outdatedness and its consequent inaccuracy.

Analysis of results. From Figure 12 it is evident that the reaction time (a measure of the overall performance of the system) strongly depends on communication delays caused by temporary disconnections. Specifically, in the vanilla case the performance is inversely proportional to the disconnection length, i.e., it decreases together with the quality of the communication links. This is in favor of a positive answer to Q1.

Also, the IRM-SA mechanisms to cope with operational uncertainty – temporary network disconnections in particular – are indeed providing a solution towards reducing the overall performance loss. Proactive self-adaptation yields smaller reaction times (Figure 12) – this is in favor of a positive answer to Q2. In particular, for the lowest critical threshold (2000ms) the reaction time is fast; this threshold configuration can, however, result into

Table 1: Two-sample tests to compare the populations of IRM-SA and control groups.

id	Alternative (Null) Hypothesis	test	median control	median IRM-SA	reject null-H?	p-value
1	The correctness of the final system architectures is lower (null: the same) in the control group than in the IRM-SA group	t-test	81.30	86.09	**Y**	0.0467
2	The IRM-SA group witnessed less (null: the same) difficulties in coming up with a DEECo architecture than the control group	Mann-Whitney	4	4	N	0.5164
3	The IRM-SA group perceived the design effort as more (null: equally) likely to be too high for an efficient use of the methodology in practice	Mann-Whitney	2	2.5	N	0.4361

Table 2: One-sample Wilcoxon Signed-Rank tests to assess a hypothesis specific to a group. Only conclusive results are shown; for the complete list of results we refer the interested reader to the experiment kit [33].

id	Alternative Hypothesis	median	p-value
4	IRM-SA group will find it easy to think of a system in IRM-SA concepts (invariants, assumptions)	3.5	0.02386
6	IRM-SA group will find IRM-SA concepts detailed enough to captured the design choices	4	0.00176
9	IRM-SA group will have increased confidence over the correctness of the architecture via IRM-SA	4	0.00731

overreactions, since it hardly tolerates any disconnections. When setting the critical threshold to 5000ms, proactive self-adaptation is triggered in case of larger disconnections (5000ms and more) only. A critical threshold of 8000ms triggers proactive self-adaptation in case of even larger disconnections (8000ms or more). Finally, when critical threshold is set to infinity, proactive self-adaptation is not triggered at all.

7.2 IRM-SA Modeling: Feasibility and Effectiveness

To evaluate the feasibility of the IRM-SA modeling process, described in section 5.3, and the impact of using IRM-SA on the effectiveness of the architects, we carried out an empirical study in the form of a controlled experiment.

7.2.1 Experiment Design

The experiment involved 20 participants: 12 Master's and 8 Ph.D. students of computer science. The participants were split into treatment (IRM-SA) and control groups. Each participant was assigned the same task, which involved coming up with a specification (on paper) of system architecture comprised of DEECo components and ensembles for a small system-to-be. The requirements of the system-to-be were provided in the form of user stories. The task's effort was comparable in size of invariants, situations and decompositions to the running example in this paper. The suggested time to accomplish the task was 4 hours, although no strict limit was imposed.

The independent variable/main factor of the experiment was the design method used: Participants in the IRM-SA group followed the IRM-SA modeling process to come up with an IRM-SA model, and then manually translated it to the system architecture, whereas participants in the control group were not recommended to use any specific method for designing the system architecture.

The dependent variables of the experiment are mapped to the hypotheses in Tables 1 and 2. We assessed (i) the correctness of system architecture [0-100] ratio scale, and (ii) several other variables capturing the intuitiveness of IRM-SA, perceived effort, adequateness of the experiment settings, etc. in Likert 5-point ordinal scale (1: Strongly disagree, 2: Disagree, 3: Not certain, 4: Agree, 5: Strongly agree). While (i) was based on manual grading each architecture based on a strict grading protocol where each error (missing knowledge, missing role, wrong process condition, wrong process period, wrong ensemble membership condition, etc.) was penalized by reducing certain number of points, assessment of (ii) was provided by the participants in pre- and post-questionnaires.

7.2.2 Results and Interpretation

To analyze the results and draw conclusions, for each hypothesis, we formulated a null hypothesis and ran a one-sided statistical test to reject it. In the case of two-sample tests (Table 1), the null hypothesis stated that the medians of the two groups were

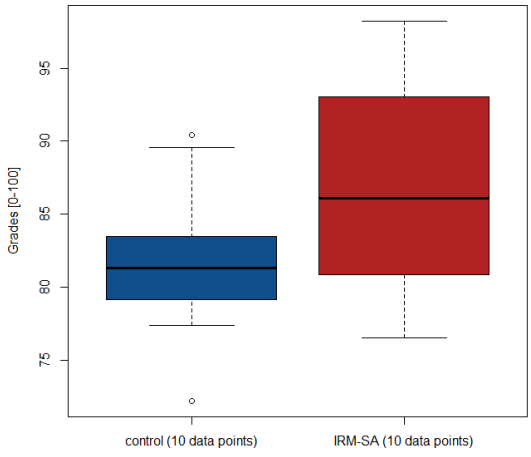


Figure 13: Box-and-whisker diagrams of the two samples measuring correctness of system architectures.

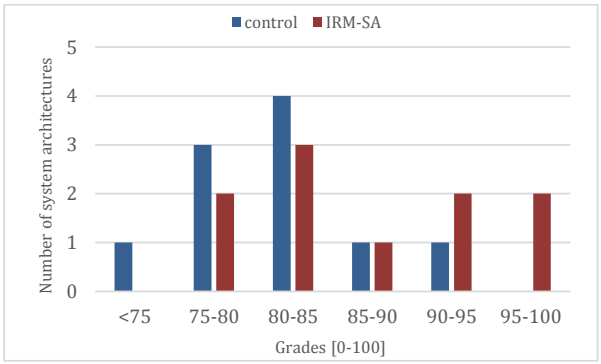


Figure 14: Histogram with distribution of grades per group.

significantly different, whereas in the case of one-sample tests (Table 2) it stated that the answer is not significantly more positive than "Not certain" (point 3 in Likert). We adopted a 5% significance level, accepting the null hypothesis if p-value<0.05. (The p-value denotes the lowest possible significance with which it is possible to reject the null hypothesis [64]).

The correctness of the system architectures showed a statistically significant difference (with p=0.0467) in favor of the IRM-SA group (Table 1, H1). Figure 13 depicts the dispersion of grades around the medians for the two groups, while Figure 14 depicts the frequencies of grades per group. From H9 (Table 2) we can also conclude that participants of the IRM-SA group perceived IRM-SA as an important factor of their confidence on the correctness of the system architecture they proposed. These two results allow us to conclude that IRM-SA increased both the actual and the perceived effectiveness of the modeling process to a statistically significant extent.

Regarding the rest of the conclusive results, participants of the IRM-SA group found the IRM-SA modeling process intuitive (H4) and the IRM-SA concepts rich enough to capture design choices (H6).

7.2.3 Threats to Validity

Conclusion validity. To perform the parametric t-test for interval/ratio data for H1, we assumed normal distribution of samples [57]. Since Likert data were treated as ordinal, we used the non-parametric Mann-Whitney tests for H2-H3 and Wilcoxon tests for H4, H6 and H9 [57]. Grading was based on a strict pre-defined protocol, which we made available together with the anonymized raw data and the replication packages in the experiment kit [33].

Internal validity. We adopted a simple "one factor with two treatments" design [64] to avoid learning effects. The number of participants (20) was high enough to reach a basic statistical validity. We used a semi-randomized assignment of participants to groups so that each group is balanced both in terms of Master's vs Ph.D. students and in terms of their experience with DEECo. Although the average time to completion of the assignment varied greatly, the mean (140 mins) and minimum (75 mins) values indicate that participants spent enough time to understand, think about and perform the task and fill in the questionnaires. The material and the experiment process was beta-tested with 4 participants beforehand.

Construct validity. We dealt with mono-method bias by using both subjective (questionnaires) and objective (grades) measures. We also measured more constructs than needed for H1 in order to obtain multiple sources of evidence. However, we introduced mono-operation bias by using only one treatment (IRM-SA) and one task. Mitigating this threat is subject of additional future studies.

External validity. Our population can be categorized as junior software engineers, since it was formed by graduate students in the last years of their studies [32]. Nevertheless, we used a rather simple example and no group assignments. Therefore, the results can be generalized mainly in academic settings.

8. DISCUSSION

8.1 Coping with Unanticipated Situations

Although IRM-SA modeling and self-adaptation as described in the previous sections relies on anticipated situations, we are aware of the fact that CPS often need to operate in situations that reside

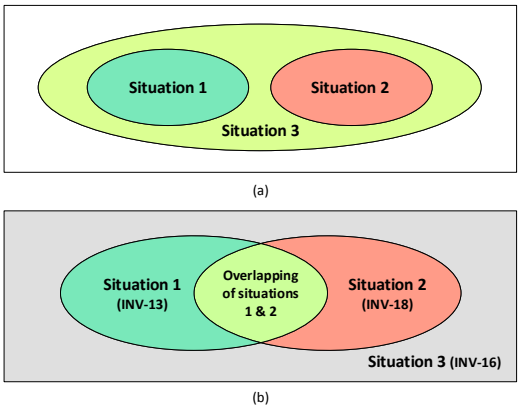


Figure 15: Overlap of situations when (a) situation 3 is a "fail-safe" mode, (b) situations overlap in the running example.

out of their "envelope of adaptability" [7]. In this section, we explain how IRM-SA tackles this problem both at runtime and design time. The driving idea is to control the decline of dependability in the system caused by unanticipated situations, so that the system's operations degrade gradually in a controlled manner.

To illustrate the problem using the running example, consider a scenario of a vegetation fire where firefighters, as a part of coordinating their actions, periodically update their group leader with information about their position as captured by personal GPS devices. A problem arises when GPS monitoring fails for whatever reasons (battery drainage etc.) – the system would no longer be able to adapt, since this failure was not anticipated at the design time. Below, we explain several strategies we propose to cope with such unanticipated situations in IRM-SA.

8.1.1 Runtime Strategy

The principal strategy for coping with unanticipated situations is to specify alternatives of OR-decompositions in such a way that they cover situations in an overlapping manner. This increases overall system robustness and fault-tolerance by providing a number of alternatives to be selected in a particular situation in the following way: when the system fails due to an unanticipated situation, there is a chance, that another alternative may be selected to preserve the top-level invariants and be unaffected by this concealed assumption.

A special case is when an IRM-SA model contains one or more design alternatives that have very weak assumptions, i.e., assumptions that are very easily satisfied at runtime, and minimum preference in an OR-decomposition. Such a design alternative is chosen only as the last option as a fail-safe mode, typical for the design of safety-critical systems. Figure 15 (a) depicts such a case, where the situation 3, reflecting a fail-safe mode, overlaps with both the situation 1 and 2.

In the running example, the runtime strategy is employed in two OR-decompositions (Figure 2). The left-most part of the decomposition of INV-9 "GM::sensorData is determined" has to be maintained differently when the associated group member is in danger (INV-13), when a nearby group member is in danger/critical state (INV-18), and when no life is in threat (INV-16). The situation characterized by INV-16 stands as a counterpart of the other two, nevertheless they are not mutually exclusive. This case is depicted in Figure 15.b.

Further, the INV-12: "GM::position is determined" has to be maintained in the two situations characterized by the INV-24: "GM indoors" and INV-26: "GM outdoors". The last two also potentially overlap, corresponding to the real-life scenario where a firefighter repeatedly enters and exits a building. In this case, the firefighter can also use the indoors tracking system to track his position; this design alternative is automatically chosen when the GPS unexpectedly malfunctions.

8.1.2 Re-design Strategy

The re-design strategy is applied in the design evolution process – occurrences of the adaptation actions that led to a failure in the system are analyzed and the IRM-SA model is revised accordingly. Such a revision can range from inclusion of a single invariant to restructuring of the whole IRM-SA model.

In such a revision, an important aid for the designer is the fact that each invariant refinement implies relationship between the sub-invariants and the parent invariant (Section 5.1). By monitoring the satisfaction of the parent invariant I_p and sub-invariants $I_1, \dots, I_n$, it is possible to narrow down the adaptation failure and infer a suitable way of addressing it. In particular, an adaptation failure occurs when:

(a) I_p is AND-decomposed, all non-process invariants among $I_1, \dots, I_n$ hold but I_p does not hold. This points to a concealed assumption in the refinement of I_p.

(b) I_p is OR-decomposed, none of its alternatives holds, but I_p holds. This points to the fact that the refinement of I_p is likely to have more strict assumptions than necessary.

(c) I_p is OR-decomposed, none of its alternatives holds, and I_p does not hold as well. This points to such an unanticipated situation, which requires either a new alternative to be introduced or an alternative that provides "close" results to be extended.

For illustration (of the case (c) in particular), consider the scenario of a non-responsive GPS. In this case, both "GM::position is determined from GPS every 1 sec" (INV-27 in Figure 2) and its parent "GM::position is determined" (INV-12) do not hold, which is a symptom for an unanticipated situation. Indeed, the root cause is that GPS signal was considered accurate. To mitigate this problem, we employ the evolution of the running example as presented in Figure 16. There, the unanticipated situation has become explicit and is used to drive the adaptation. Specifically, in this new situation, the system still satisfies INV-12 by switching to the right-most alternative. In such a case, the Firefighter's position is determined by aggregating the positions of the nearby firefighters (INV-36) and estimating its own position based on these positions and the radius of search, through determining the maximum overlapping area (INV-37).

8.2 Self-Adaptation in Decentralized Settings

In decentralized settings, each node has up-to-date enough data and solves independently the same SAT problem. Communication delays may however cause temporary de-synchronization of components' knowledge and, therefore, components may reach different solutions. While this de-synchronization decreases the overall performance of the system, it is not harmful in major cases, as long as the system has revertible and gradual responses.

To assess the harmfulness of such de-synchronization we expect the existence of a relative utility function $r(\Delta t_{\max})$, which measures the ratio of utilities: (i) where the SAT solving is based on values that are as much old as maximum communication delays expected in the correct functioning of the system, and (ii)

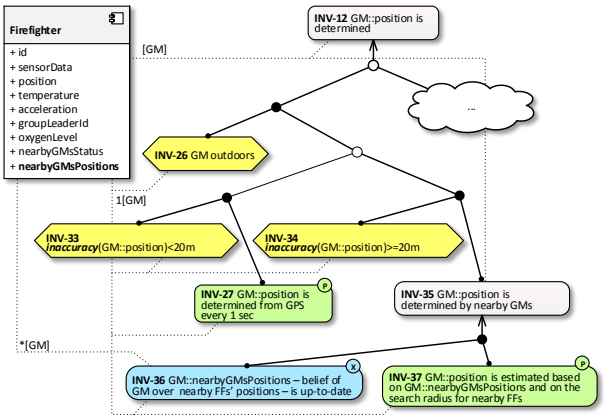


Figure 16: Design evolution scenario – the new "*inaccuracy*(GM::position) < 20m" situation is added to the model and drives the adaptation.

the ideal case of SAT solving having instantaneous access to the freshest values. The exact notion of the relative utility function $r(\Delta t_{\max})$ is formalized in Appendix A.

There are several factors the influence the value of $r(\Delta t_{\max})$. Essentially, this depends on the shape of the utility function and (as the utility function is cumulative), on the duration of situations when the system is de-synchronized (*divergence*). During divergence, some nodes of the system arrive at different solutions due to their differently outdated knowledge. (Formally, this corresponds to condition $a_{c,d}(t) \neq a(t)$ – see Appendix A.) More specifically, it means that there are assumptions in the IRM-SA model that one component considers satisfied, while another considers violated. Following this argument, we identify three categories of system properties that collectively contribute to the value of $r(\Delta t_{\max})$ and characterize the ability of the system to work well with the decentralized SAT solving employed in IRM-SA:

Criticality of a particular system operation affected by divergence. For critical operations, the utility function tends to get extreme negative values, thus even a short operation under divergence yields very low overall utility. On the other hand if the environment has revertible and gradual responses, it hardly matters whether the system is in divergence for a limited time (e.g., if the system controls a firefighter walking around a building, then walking in a wrong direction for a few seconds does not cause any harm and its effect can be easily reverted).

Dynamicity of process in the environment. As the system reacts to changes in the environment, it is impacted by the speed these changes happen. Such a change creates potential divergence in the system. What matters then is the ratio between the time needed to converge after the divergence and the interval between two consecutive changes. For instance, if house numbers were to be observed by firefighters when passaging by, then walking speed would yield much slower rate of changes then passing by in a car.

Sensitivity of assumptions. This is a complementary aspect of the previous property**.** Depending on the way assumptions are refined into computable ones, one can detect fine-grained changes in the environment. For example, consider an assumption that relies on computing position changes of a firefighter moving in a city. Computing position changes based on changes in the house number obviously yields more frequent (observable) changes in

the environment than computing changes based on the current street number.

8.3 Applicability of IRM-SA beyond DEECo

The IRM-SA concepts of design components and invariants, and the process of elaborating invariants via AND- and OR-decomposition are implementation-agnostic. The outcome therefore of the design – the IRM-SA model – can be mapped to a general-purpose (e.g., OSGi [27], Fractal [9], SOFA 2 [12]) or real-time component model (e.g., RTT [61], ProCom [56]) that provides explicit support for modeling entities that encapsulate computation (components) and interaction (connectors). In a simple case, the mapping involves translating the process and exchange invariants to local component activities and to connectors, respectively.

That said, the mapping of IRM-SA to DEECo is particularly smooth. This is because IRM-SA is tailored to the domain of dynamic self-adaptive CPS, where the management of belief of individual components via soft-real-time operation and dynamic grouping has a central role. In this regard, component models tailored to self-adaptive CPS (e.g., DEECo) and architectural styles for group-based coordination (e.g., A-3 [3]), are good candidates for a smooth mapping, since they provide direct implementation-level counterparts of the IRM-SA abstractions.

9. RELATED WORK

Recently, there has been a growing interest in software engineering research for software-intensive systems with self-adaptive and autonomic capabilities [53]. Since our approach is especially close to software product lines approaches and their dynamic counterparts, we first position IRM-SA against these approaches; then we split the comparison into three essential views of self-adaptation [14], namely requirements, assurances, and engineering.

9.1 IRM-SA vs. Dynamic Software Product Lines

Variability modeling at design time has been in the core of research in software product lines (SPLs) [17]. Recently, numerous approaches have proposed to apply the proven SPL techniques of managing variability in the area of runtime adaptation, leading to the concept of dynamic SPL (DSPL) [29]. The idea behind DSPL is to delay the binding of variants from design to runtime, to focus on a single product than a family of products, and to have configurations be driven by changes in the environment than by market needs and stakeholders. In this spirit, IRM-SA can be considered as a DSPL instance.

Within DSPLs, feature models are an effective means to model the common features, the variants, and the variability points of a self-adapting system [35]. However, they fall short in identifying and modeling the situations that the system may encounter during operation [55]. This results in maintaining an additional artifact as a context model (e.g., an ontology [49]) in parallel with the feature model at runtime, in order to bind the monitoring and planning phases of the MAPE-K loop [36]. IRM-SA provides an explicit support for capturing situations, pertinent to system operation and self-adaptation, via the assumption concept. At the same time, by building on the iterative refinement of invariants and the assignment of leaf invariants to design components, IRM-SA integrates the problem space view (requirements models) with the solution space view (feature models) into a single manageable artifact (IRM-SA model).

Finally, compared to existing approaches in DSPLs that use goal models to describe the configuration space [55], IRM-SA does not focus on the prioritization between applicable configurations, but on the gradual degradation of overall system performance when dealing with operational uncertainty in decentralized settings.

9.2 Requirements and Assurances

In an effort to study the requirements that lead to the feedback loop functionality of adaptive systems, Souza et al. defined a new class of requirements, termed “awareness requirements” [59], which refer to the runtime success/failure/quality-of-service of other requirements (or domain assumptions). Awareness requirements are expressed in an OCL-like language, based on the Tropos goal models [8] produced at design time, and monitored at runtime by the requirements monitoring framework ReqMon [52]. The idea is to have a highly sophisticated logging framework, on top of which a full MAPE-K feedback loop [36] can be instantiated. Our approach, on the other hand, features a tighter coupling between monitoring and actuating, since both aspects are captured in the IRM-SA model.

Extending goal-based requirements models with alternative decompositions to achieve self-adaptivity has been carried out in the frame of Tropos4AS [45, 46]. System agents, together with their goals and their environment are first modeled and then mapped to agent-based implementation in Jadex platform. Although IRM-SA is technically similar to Tropos4AS, it does not capture the goals and intentions of individual actors, but the desired operation of the system as a whole, thus promoting dependable operation, key factor in CPS.

For dealing with uncertainty in the requirements of self-adaptive systems, the RELAX [63] and FLAGS [4] languages have been proposed. RELAX syntax is in the form of structured natural language with Boolean expressions; its semantics is defined in a fuzzy temporal logic. The RELAX approach distinguishes between invariant and non-invariant requirements, i.e., requirements that may not have to be satisfied at all times. In [15], RELAX specifications are integrated into KAOS goal models. Threat modeling à la KAOS [42] is employed to systematically explore (and subsequently mitigate) uncertainty factors that may impact the requirements of a DAS. FLAGS is an extension to the classical KAOS goal model that features both crisp goals (whose satisfaction is Boolean) and fuzzy goals (whose satisfaction is represented by fuzzy constrains) that correspond to invariant and non-invariant requirements of RELAX. The idea of using a FLAGS model at runtime to guide system adaptation has been briefly sketched in [4], but to the best of our knowledge not yet pursued. Compared to IRM-SA, both RELAX and FLAGS focus on the requirements domain and do not go beyond goals and requirements to design and implementation.

One of the first attempts to bind requirements, captured in KAOS, with runtime monitoring and reconciliation tactics is found in the seminal work of Fickas and Feather [21, 22]. Their approach is based on capturing alternative designs and their underlying assumptions via goal decomposition, translating them into runtime monitors, and using them to enact runtime adaptation. Specifically, breakable KAOS assumptions (captured in LTL) are translated into the FLEA language, which provides constructs for expressing a temporal combination of events. When requirements violation events occur, corrective actions apply, taking the form of either parameter tuning or shifting to alternative designs. There are two main differences to our approach: (i) in KAOS-FLEA the designer has to manually write and tune the reconciliation tactics, whereas we rely on the structure of an IRM-SA model and the

solver for a solution; (ii) contrary to KAOS-FLEA we do not treat alternative designs as correcting measures, but as different system modes, which is more suitable for CPS environments.

9.3 Design and Implementation

At the system design and implementation phases, the component-based architectural approach towards self-adaptivity is favored in several works [23, 51]. Here, mode switching stands as a widely accepted enabling mechanism, introduced by Kramer and Magee [30, 39]. The main shortcoming is that mode switching is triggered via a finite state machine with pre-defined triggering conditions, which is difficult to trace back to system requirements. Also, although partially addressed in [48], modes and the triggering conditions are usually designed via explicit enumeration, which may cause scalability problems given the number and complex mutual relations of the variation points involved. In IRM-SA, architecture configurations act as modes. However, IRM-SA complements mode switching by enabling for compositional definition of architecture configurations and providing the traceability links, which in turn allows for self-explanation [54].

Another large area of related work focusing specifically on decentralized operation to achieve a common joint goal is found in distributed and multi-agent planning. Here the main questions are centered around the issue of how to compute a close-to-optimal plan for decentralized operation of agents with partially observable domains. The typical solution is to model the environment of an agent via a Decentralized Partially Observable Markov Decision Process [6]. This is further extended to include imperfect communication [60, 66] and taken even a step further, when decentralization is strengthened by not requiring prior coordination (and strict synchronization) – either by resigning on the inter-agent communication [5, 65] or by performing communication simultaneously to planning [62]. The major difference to our work is that multi-agent planning requires a model of the environment in order to predict the effect of an action. A complete model of an environment becomes a rather infeasible requirement. IRM-SA does not require the complete model of an environment as the plan is not computed, but specified by a designer. By including high-level invariants, the IRM-SA however still able to reason about the efficiency of the configuration being currently executed and it also drives decentralized decisions on the configuration selection.

Finally, architecture adaptation based on various constraint solving techniques is not a new idea. A common conceptualization, e.g., in [24, 28], is based on the formal definition of architecture constraints (e.g., architecture styles, extra-functional properties), individual architecture elements (e.g., components), and, most importantly, adaptation operations that are supported by an underlying mechanism (e.g., addition/removal of a component binding). Typically, the objective is to find an adaptation operation that would bring the architecture from an invalid state to a state that conforms to the architecture constraints (architecture planning). Although these methods support potentially unbounded architecture evolution (since they are based on the supported adaptation operations rather than predefined architecture configurations), they typically consider only structural and extra-functional properties rather than system goals. Consequently, they support neither smooth, gradual degradation in unanticipated situations, nor offline evolution of the design.

10. CONCLUSION AND FUTURE WORK

In this paper, we have presented the IRM-SA method – an extension to IRM that allows designing of self-adaptive Cyber-Physical Systems (CPS) with a focus on dependability aspects. The core idea of the method is to describe variability of a system by alternative invariant decompositions and then to drive system adaptation by employing the knowledge of high-level system's goals and their refinement to computational activities.

A novel feature of IRM-SA is that it allows adaptation in presence of operational uncertainty caused by inaccuracies of sensed data and by unreliable communication. This is achieved by reasoning not only on the values of belief on component knowledge, but also on the degree of belief inaccuracy, which is based on the degree of belief outdatedness and the model that governs the evolution of belief.

As a proof of concept, we have implemented IRM-SA within jDEECo (a Java realization of the DEECo component model) and showed its feasibility by modeling a real-life scenario from an emergency coordination case study. Moreover, we have provided a proof-of-concept demonstration of the benefits of IRM-SA by experiments carried out by simulation. We have also tested the feasibility and effectiveness of the IRM-SA modeling process via a controlled experiment.

Future work. When the SAT solver fails to find an applicable configuration, self-adaptation based on IRM-SA reaches its limits. In this case, a promising idea is to rely on machine learning and other AI techniques to provide complementary strategies for the system dynamically extend its envelope of adaptability [25]. Investigating the concrete impact of applying these strategies in the frame of IRM-SA and coming up with new strategies is a subject of our current and future work.

11. ACKNOWLEDGEMENTS

This work was partially supported by the EU project ASCENS 257414 and by Charles University institutional funding SVV-2015-260222. The research leading to these results has received funding from the European Union Seventh Framework Programme FP7-PEOPLE-2010-ITN under grant agreement n°264840.

12. REFERENCES

[1] Ali, R. Al, Bures, T., Gerostathopoulos, I., Keznikl, J. and Plasil, F. 2014. Architecture Adaptation Based on Belief Inaccuracy Estimation. *Proc. of WICSA'14* (Sydney, Australia, Apr. 2014), 1–4.

[2] Arcaini, P., Gargantini, A. and Vavassori, P. 2015. Generating Tests for Detecting Faults in Feature Models. *Proc. of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST '15), to appear.* (Apr. 2015).

[3] Baresi, L. and Guinea, S. 2011. A-3: An Architectural Style for Coordinating Distributed Components. *2011 9th Working IEEE/IFIP Conference on Software Architecture (WICSA)* (Jun. 2011), 161–170.

[4] Baresi, L., Pasquale, L. and Spoletini, P. 2010. Fuzzy Goals for Requirements-Driven Adaptation. *2010 18th IEEE International Requirements Engineering Conference*. (Sep. 2010), 125–134.

[5] Barrett, S., Stone, P., Kraus, S. and Rosenfeld, A. 2013. Teamwork with Limited Knowledge of Teammates. *Proceedings of the 27th AAAI Conference on Artificial Intelligence* (Jul. 2013).

[6] Bernstein, D.S., Givan, R., Immerman, N. and Zilberstein, S. 2002. The Complexity of Decentralized Control of Markov

Decision Processes. *Math. Oper. Res.* 27, 4 (Nov. 2002), 819–840.

[7] Berry, D.M., Cheng, B.H.C. and Zhang, J. 2005. The Four Levels of Requirements Engineering for and in Dynamic Adaptive Systems. *Proc. of the 11th International Workshop on Requirements Engineering Foundation for Software Quality, Porto, Portugal* (2005), 95–100.

[8] Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F. and Mylopoulos, J. 2004. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*. 8, 3 (May 2004), 203–236.

[9] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V. and Stefani, J.-B. 2006. The Fractal component model and its support in Java. *Software: Practice & Experience*. 36, 11-12 (2006), 1257–1284.

[10] Bures, T., Gerostathopoulos, I., Hnetynka, P. and Keznikl, J. 2014. Gossiping Components for Cyber-Physical Systems. *Proc. of 8th European Conference on Software Architecture* (2014), 250–266.

[11] Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M. and Plasil, F. 2013. DEECo – an Ensemble-Based Component System. *Proc. of CBSE'13* (Vancouver, Canada, Jun. 2013), 81–90.

[12] Bures, T., Hnetynka, P. and Plasil, F. 2006. SOFA 2.0 : Balancing Advanced Features in a Hierarchical Component Model. *SERA '06* (2006), 40–48.

[13] Chang, N., Rashidzadeh, R. and Ahmadi, M. 2010. Robust indoor positioning using differential wi-fi access points. *IEEE Transactions on Consumer Electronics*. 56, 3 (Aug. 2010), 1860–1867.

[14] Cheng, B. et al. 2009. Software Engineering for Self-Adaptive Systems: A Research Roadmap. *Software Engineering for Self-Adaptive Systems*. Springer Berlin Heidelberg. 1–26.

[15] Cheng, B.H.C., Sawyer, P., Bencomo, N. and Whittle, J. 2009. A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. *Proc. of the 12th International Conference on Model Driven Engineering Languages and Systems, MoDELS '09* (2009), 1–15.

[16] Chung, L., Nixon, B., Yu, E. and Mylopoulos, J. 1999. *Non-Functional Requirements in Software Engineering*. Springer.

[17] Clements, P. and Northrop, L. 2002. *Software Product Lines: Practices and Patterns*. Addison Wesley Professional.

[18] Dalpiaz, F., Chopra, A.K., Giorgini, P. and Mylopoulos, J. 2010. Adaptation in Open Systems: Giving Interaction its Rightful Place. *Proceedings of the 29th International Conference on Conceptual Modeling (ER '10)* (Vancouver, Canada, Nov. 2010), 31–45.

[19] DeNicola, R., Ferrari, G., Loreti, M. and Pugliese, R. 2013. A Language-based Approach to Autonomic Computing. *Formal Methods for Components and Objects*. 7542, (2013), 25–48.

[20] Esfahani, N., Kouroshfar, E. and Malek, S. 2011. Taming uncertainty in self-adaptive software. *Proc. of SIGSOFT/FSE '11* (2011), 234–244.

[21] Feather, M.S., Fickas, S., van Lamsweerde, A. and Ponsard, C. 1998. Reconciling System Requirements and Runtime Behavior. *Proceedings of the 9th International Workshop on Software Specification and Design* (1998), 50–59.

[22] Fickas, S. and Feather, M.S. 1995. Requirements monitoring in dynamic environments. *Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE'95)*. (1995), 140–147.

[23] Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B. and Steenkiste, P. 2004. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*. 37, 10 (2004), 46–54.

[24] Georgiadis, I., Magee, J. and Kramer, J. 2002. Self-organising software architectures for distributed systems. *Proceedings of the first workshop on Self-healing systems - WOSS '02* (2002), 33–38.

[25] Gerostathopoulos, I., Bures, T., Hnetynka, P., Hujecek, A., Plasil, F. and Skoda, D. *Meta-Adaptation Strategies for Adaptation in Cyber- Physical Systems*. Technical Report #D3S-TR-2015-01, April 2015. Department of Distributed and Dependable Systems. Available at: http://d3s.mff.cuni.cz/publications/.

[26] Goldsby, H.J., Sawyer, P., Bencomo, N., Cheng, B.H.C. and Hughes, D. 2008. Goal-Based Modeling of Dynamically Adaptive System Requirements. *Proc. of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2008)* (Mar. 2008), 36–45.

[27] Hall, R., Pauls, K., McCulloch, S. and Savage, D. 2011. *OSGi in Action: Creating Modular Applications in Java*. Manning Publications, Stamford, CT.

[28] Hansen, K.M. 2012. Modeling and Analyzing Architectural Change with Alloy. *SAC '10* (2012), 2257–2264.

[29] Hinchey, M., Park, S. and Schmid, K. 2012. Building Dynamic Software Product Lines. *Computer*. 45, 10 (Oct. 2012), 22–26.

[30] Hirsch, D., Kramer, J., Magee, J. and Uchitel, S. 2006. Modes for software architectures. *Proc. of the 3rd European conference on Software Architecture, EWSA '06* (2006), 113–126.

[31] Hoelzl, M., Rauschmayer, A. and Wirsing, M. 2008. Engineering of Software-Intensive Systems: State of the Art and Research Challenges. *Software-Intensive Systems and New Computing Paradigms*. 1–44.

[32] Höst, M., Regnell, B. and Wohlin, C. 2000. Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Software Engineering*. 214, (2000), 201–214.

[33] IRM-SA Website: 2015. *http://d3s.mff.cuni.cz/projects/irm-sa*. Accessed: 2015-04-23.

[34] jDEECo Website: 2015. *https://github.com/d3scomp/JDEECo*. Accessed: 2015-04-23.

[35] Kang, K.C., Jaejoon, L. and Donohoe, P. 2002. Feature-oriented product line engineering. *IEEE Software*. 19, 4 (2002), 58–65.

[36] Kephart, J. and Chess, D. 2003. The Vision of Autonomic Computing. *Computer*. 36, 1 (2003), 41–50.

[37] Keznikl, J., Bures, T., Plasil, F., Gerostathopoulos, I., Hnetynka, P. and Hoch, N. 2013. Design of Ensemble-Based Component Systems by Invariant Refinement. *Proc. of CBSE'13* (Vancouver, Canada, Jun. 2013), 91–100.

[38] Keznikl, J., Bures, T., Plasil, F. and Kit, M. 2012. Towards Dependable Emergent Ensembles of Components: The

DEECo Component Model. *Proc. of WICSA/ECSA'12* (Aug. 2012), 249–252.

[39] Kramer, J. and Magee, J. 2007. Self-managed systems: an architectural challenge. *Proc. of FOSE'07* (Minneapolis, USA, May 2007), 259–268.

[40] Lamsweerde, A. Van 2008. Requirements engineering: from craft to discipline. *16th ACM Sigsoft Intl. Symposium on the Foundations of Software Engineering* (Atlanta, USA, Nov. 2008), 238–249.

[41] Lamsweerde, A. Van 2009. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. John Wiley and Sons.

[42] Lamsweerde, A. Van and Letier, E. 2000. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering*. 26, 10 (2000), 978–1005.

[43] LeBerre, D. and Parrain, A. 2010. The Sat4j Library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*. 7, (2010), 59–64.

[44] McKinley, P.K., Sadjadi, S.M., Kasten, E.P. and Cheng, B.H.C. 2004. Composing adaptive software. *Computer*. 37, 7 (2004), 56–64.

[45] Morandini, M., Penserini, L. and Perini, A. 2008. Automated Mapping from Goal Models to Self-Adaptive Systems. *2008 23rd IEEE/ACM International Conference on Automated Software Engineering* (Sep. 2008), 485–486.

[46] Morandini, M. and Perini, A. 2008. Towards Goal-Oriented Development of Self-Adaptive Systems. *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems - SEAMS '08* (Leipzig, Germany, May 2008), 9–16.

[47] Morin, B., Barais, O., Jezequel, J.-M., Fleurey, F. and Solberg, A. 2009. Models at Runtime to Support Dynamic Adaptation. *Computer*. 42, 10 (2009), 44–51.

[48] Morin, B., Barais, O., Nain, G. and Jezequel, J. 2009. Taming Dynamically Adaptive Systems Using Models and Aspects. *Proc. of the 31st International Conference in Software Engineering, ICSE '09* (2009), 122–132.

[49] Murguzur, A., Capilla, R., Trujillo, S., Ortiz, Ó. and Lopez-Herrejon, R.E. 2014. Context Variability Modeling for Runtime Configuration of Service-based Dynamic Software Product Lines. *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2* (New York, NY, USA, 2014), 2–9.

[50] Murray, R.M., Astrom, K.J., Boyd, S.P., Brockett, R.W. and Stein, G. 2003. Future Directions in Control in an Information-Rich World. *Control Systems, IEEE*. 23, 2 (2003), 1–21.

[51] Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S. and Wolf, A.L. 1999. An Architecture-Based Approach to Self-Adaptive Software. *Intelligent Systems and their Applications, IEEE*. 14, 3 (1999), 54 – 62.

[52] Robinson, W.N. 2005. A requirements monitoring framework for enterprise systems. *Requirements Engineering*. 11, 1 (Nov. 2005), 17–41.

[53] Salehie, M. and Tahvildari, L. 2009. Self-Adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems*. 4, 2, May (2009), 1–40.

[54] Sawyer, P., Bencomo, N., Whittle, J., Letier, E. and Finkelstein, A. 2010. Requirements-Aware Systems: A Research Agenda for RE for Self-adaptive Systems. *Proc. of the 18th IEEE International Requirements Engineering Conference* (Sep. 2010), 95–103.

[55] Sawyer, P., Rocquencourt, I., Mazo, R., Diaz, D., Salinesi, C. and Paris, U. 2012. Using Constraint Programming to Manage Configurations in Self-Adaptive Systems. *Computer*. 45, 10 (2012), 56–63.

[56] Sentilles, S., Vulgarakis, A., Bures, T., Carlson, J. and Crnkovic, I. 2008. A component model for control-intensive distributed embedded systems. *Proc. of the 11th International Symposium on Component-Based Software Engineering* (Oct. 2008), 310–317.

[57] Sheskin, D.J. 2011. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman and Hall/CRC.

[58] Shoham, Y. and Leyton-Brown, K. 2008. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press.

[59] Souza, V.E.S., Lapouchnian, A., Robinson, W.N. and Mylopoulos, J. 2013. Awareness Requirements. *Software Engineering for Self-Adaptive Systems II*. Springer Berlin Heidelberg. 133–161.

[60] Spaan, M.T.J., Oliehoek, F.A. and Vlassis, N. 2008. Multiagent Planning under Uncertainty with Stochastic Communication Delays. *Proc. of Int. Conf. on Automated Planning and Scheduling* (2008), 338–345.

[61] The Orocos Real-Time Toolkit: *http://www.orocos.org/wiki/orocos/rtt-wiki*.

[62] Valtazanos, A. and Steedman, M. 2014. Improving Uncoordinated Collaboration in Partially Observable Domains with Imperfect Simultaneous Action Communication. *Proc. of the Workshop on Distributed and Multi-Agent Planning in ICAPS* (2014), 45–54.

[63] Whittle, J., Sawyer, P. and Bencomo, N. 2010. RELAX: A Language to Address Uncertainty in Self-Adaptive Systems Requirements. *Requirements Engineering*. 15, 2 (2010), 177–196.

[64] Wohlin, C., Runeson, P., Host, M., Ohlsson, M.C., Regnell, B. and Wesslen, A. 2012. *Experimentation in Software Engineering*. Springer.

[65] Wu, F., Zilberstein, S. and Chen, X. 2011. Online Planning for Ad Hoc Autonomous Agent Teams. *Proc. of the 22nd International Joint Conference on Artificial Intelligence* (2011), 439–445.

[66] Wu, F., Zilberstein, S. and Chen, X. 2011. Online planning for multi-agent systems with bounded communication. *Artificial Intelligence*. 175, 2 (Feb. 2011), 487–511.

Appendix A – Formalization of System Utility

This appendix provides formalization of the relative system utility and related assumptions, as used in Section 8.2.

Definition 1 (Utility of a system run). For a given run r of a system:

- Let $a(t)$ be a function that for time t returns the set of active processes as selected by the SAT procedure.
- Let $d(t,k)$ be function that for time t and a knowledge field k returns the time Δt that has elapsed since the knowledge value contained in k was sensed/created by the component where the knowledge value originated from.
- Let $a_{c,d}(t)$ be a function that for given time t returns a set of active processes of component c as selected by the SAT procedure assuming that knowledge values outdated by $d(t,k)$ have been used. Further, we denote $a_{d_1,\dots,d_n}(t) = \bigcup_i a_{c_i,d_i}(t)$ as the combination over components existing in the system. (In other words, each component selects active processes itself based on its belief, which is differently outdated for each component.)
- Let $u(a)$ be a cumulative utility function that returns the overall system utility when performing processes $a(t)$ at each time instant t.
- Let $u(a,\Delta t_{\max})$ be a cumulative utility function constructed as $\min\{u(a_{d_1,\dots,d_n}) \mid d(k) \le \Delta t_{\max}\}$. (In other words, $u(a,\Delta t_{\max})$ denotes the lowest utility if components decide independently based on knowledge at most $\Delta t_{\max}$ old.)

Definition 2 (Expected relative utility). Let $\mathrm{E}(u(a))$ be the expected value of $u(a)$ and $\mathrm{E}(u(a,\Delta t_{\max}))$ be the expected value of $u(a,\Delta t_{\max})$. Assuming that $\mathrm{E}(u(a)) > 0$ (i.e. in the ideal case of zero communication delays the system provides a positive value), we define expected relative utility as $r(\Delta t_{\max}) = \mathrm{E}(u(a,\Delta t_{\max}))/\mathrm{E}(u(a))$.

In the frame of these definitions, we assume systems where $r(\Delta t_{\max})$ is close to 1 (and definitely non-negative) for given upper bound on communication delays $\Delta t_{\max}$. In fact $r(\Delta t_{\max})$ provides a continuous measure of how well the method works in a distributed environment.

Considering that the communication in the system happens periodically and that an arriving message obsoletes all previous not-yet-arrived messages from the same source, $\Delta t_{\max}$ can be set to $q_l + q_d T$, where q_l is a close-to-100% quantile of the distribution of message latencies, T is the period of message sending and q_d is a close-to-100% quantile of the distribution of the length of sequences of consecutive message drops. Naturally, if there is a chance of some (not necessarily reliable communication), $\Delta t_{\max}$ can be set relatively low while still covering the absolute majority of situations.

4.7 A Life Cycle for the Development of Autonomic Systems: The e-Mobility Showcase

Tomáš Bureš,
Rocco De Nicola,
Ilias Gerostathopoulos,
Nicklas Hoch,
Michal Kit,
Nora Koch,
Giacoma Valentina Monreale,
Ugo Montanari,
Rosario Pugliese
Nikola Serbedzija,
Martin Wirsing,
Franco Zambonelli

In proceedings of the **7th IEEE International Conference on Self-Adaptation and Self-Organizing Systems Workshops (SASOW 2013)**.

Published by IEEE,
pages 71-76,
ISBN 978-1-4799-5086-7,
September 2013.

The original version is available electronically from the publisher's site at http://dx.doi.org/10.1109/SASOW.2013.23.

Summary of the Paper

This paper, published as [BDNG+13], positions the work performed in this thesis, in particular the proposed *Invariant Refinement Method* (IRM), within the development of autonomic systems as featured by the EU project ASCENS [1]. The focus of this paper, and of ASCENS, is on software engineering of large distributed and dynamic systems akin to software-intensive Cyber-Physical Systems (siCPS). The concept of *component ensembles* (Section 2.2.2) is put forward as a promising way to engineer such systems.

The main idea of the paper is to introduce a methodological model that guides the development of ensemble-based systems and exemplify it on the ASCENS case study of electric vehicle navigation. The model, called *ensemble development life cycle* (EDCL) comprises an offline and an online feedback loop (referred to as "double-wheel" model), each comprising three activities [HKP+15]. The offline loop features the design activities of requirements engineering, modeling and programing, and verification and validation, whereas the online loop features the runtime activities of monitoring, awareness, and self-adaptation (being comparable to the MAPE-K loop for autonomic systems [KC03]). These two loops are connected via the activities of deployment and feedback.

From the perspective of the EDLC, IRM is a method that provides a transition from the requirements engineering phase to the modeling and programing phase. In particular, IRM provides a mechanism to translate the requirements of an ensemble-based system, expressed as high-level invariants, to low-level obligations of components and component ensembles. It thus can be used for the high-level (architecture) design phase (which is not explicitly represented in the double-wheel EDLC model). Within the EDLC, IRM receives input from SOTA [ABZ12], a requirements engineering method, and provides its output to SCEL [DNFLP13], a language for process modeling inspired by process algebra formalisms.

Although this paper focuses on IRM and does not take into account IRM's extension for self-adaptivity (IRM-SA), its results apply also to IRM-SA: since IRM-SA is an extension of IRM, it can also be used in the architecture design phase of the EDLC.

Comments on Authorship

Overall, the paper and its main idea of the ensemble development lifecycle are of equal authorship and reflect the collaboration among several partners participating in the ASCENS project [1]. I was personally focusing on the high-level architecture design phase of the life cycle, where I contributed in proposing IRM as a possible design method and elaborated on how it can be employed to model the running example of cooperative e-vehicles.

A Life Cycle for the Development of Autonomic Systems: The e-Mobility Showcase

Tomas Bures[1], Rocco De Nicola[2], Ilias Gerostathopoulos[1], Nicklas Hoch[3], Michal Kit[1],
Nora Koch[4], Giacoma Valentina Monreale[5], Ugo Montanari[5], Rosario Pugliese[6], Nikola Serbedzija[7],
Martin Wirsing[4], Franco Zambonelli[8]
[1]Charles University in Prague, Faculty of Mathematics and Physics, Czech Republic
[2]IMT Institute for Advanced Studies Lucca, Italy
[3]Corporate Research Group, Volkswagen AG, Wolfsburg, Germany
[4]Ludwig-Maximilians-Universität München, Germany [5]Dipartimento di Informatica, Università di Pisa, Italy
[6]Università di Firenze, Italy [7]Fraunhofer FOKUS Berlin, Germany [8]Università di Modena e Reggio Emilia, Italy

***Abstract*—Component ensembles are a promising way of building self-aware autonomic adaptive systems. This approach has been promoted by the EU project ASCENS, which develops the core idea of ensembles by providing rigorous semantics as well as models and methods for the whole development life cycle of an ensemble-based system. These methods specifically address adaptation, self-awareness, self-optimization, and continuous system evolution. In this paper, we demonstrate the key concepts and benefits of the ASCENS approach in the context of intelligent navigation of electric vehicles (e-Mobility), which itself is one of the three key case studies of the project.**

***Keywords*—autonomic systems; life cycle; ensembles;**

I. Introduction

The development of massively distributed and highly dynamic systems which interact with and control the physical world is one of the major challenges in software engineering [1]. This is because the dynamicity of these systems and the not well foreseeable context brought by the external physical environment demand that software operating in these systems is highly self-aware, autonomic and adaptive. While self-awareness and adaptivity has been relatively well mastered in case of small-scale localized control (especially in the field of control systems), it is still a major problem for large-scale distributed systems, which are open-ended and dynamic (meaning that components of the system may freely appear and disappear as well as change their communication partners).

Within the EU project ASCENS[1] an approach based on ensembles of components is pursued. Contrary to classical component-based software engineering, it features important concepts of knowledge and ensembles. The *knowledge* of a component is a structured repository of facts with well-defined relations. The facts in the knowledge change at runtime to reflect the state of the component and its belief about its environment, thus effectively addressing the self-awareness of a component. The *ensembles* are dynamic goal-oriented communication groups of components. The ensembles are formed on demand to reflect intentions of components with respect to the current state of their environment. This way, ensembles address the dynamicity and adaptivity of components.

In addition to providing basic concepts and their semantics, ASCENS wraps this into a holistic *ensemble development life cycle* (EDLC) framework, which covers the full development life cycle and addresses design and development for adaptation, self-awareness, self-optimization, and continuous system evolution.

In this paper we take a practitioner's approach and demonstrate the application of the EDLC on the development of one of the key ASCENS case-studies – the intelligent navigation of electric vehicles (e-Mobility).

The paper is structured as follows: Section II describes the e-Mobility case study and Section III outlines the EDLC and describes a high-level strategy of applying it to e-Mobility. Sections IV–IX demonstrate the particular EDCL steps applied. The evaluation and related work is presented in Section X, while Section XI concludes the paper.

II. E-Mobility Case Study

The e-Mobility scenario focuses on avoiding contingency situation in an open-ended systems of interacting electric vehicles. Such a scenario is highly dynamic. This stems mostly from the fact that it includes unforseeable human user actions which influence the availability of travel resources.

Technically, we assume in the case-study that travels are initiated by personal activities. A journey is thus defined as a sequence of trips, with each trip being initiated by a single activity. Trips may consist of multiple stages. A stage can be executed in different travel modes such as walking mode or driving mode. For example, consider a user that leaves for work in the morning. Work is the activity that initiates travel. The first trip contains a walking stage from home to the vehicle's parking lot, a driving stage from the parking lot at home to the one at work and lastly a walking stage to the office. The working time at the office is considered to be the activity duration. Throughout that time the vehicle is parked at the car park. If it has access to a charging station, it may recharge.

This work has been sponsored by the EU project ASCENS, FP7 257414.
[1]http://www.ascens-ist.eu

After work the user continues his journey. The number of consecutive trips follows from the number of activities.

In this scenario the main components are the user, the electric vehicle, the parking lot and charging station. Parking lot and charging station are commonly referred to as infrastructure components. Component temporarily form ensembles. These ensembles include (i) collection of charging stations, (ii) collection of parking lots, (iii) collection of users and electric vehicles and (iv) collection of at least one user, one electric vehicle and one infrastructure component, etc.

Throughout runtime, contingency situations may occur. Components and ensembles require self-adaptive actions to resolve these situations. Examples of contingency situations that need to be resolved by the electric vehicle component include (i) unavailability of a reserved parking lot, (ii) unavailability of a reserved charging station, (iii) falling below minimum battery energy level and (iv) missing a scheduled arrival time. Examples of contingency situations that need to be handled by the parking lot or charging station component include (i) early or late arrival of a vehicle at a parking lot or charging station, (ii) early or late departure of a vehicle from a parking lot or charging station, (iii) missed initiation of a scheduled charging action and (iv) deviation from the expected power profile during charging.

III. Applying EDLC to e-Mobility – Big Picture

Within the scope of the ASCENS project we propose a "double-wheel" *ensemble development life cycle* (EDLC) – see Fig. 1 – for autonomic systems such as the e-Mobility. The aim is to provide a conceptual framework that covers the main aspects of the engineering process required for such systems. The "first wheel" is used for representing the phases that are performed *offline*, which are mainly those related to *design*. The "second wheel" focus on the phases related to *online* activities that are performed at *runtime*. Both are connected by the transitions *deployment* and *feedback* from design to runtime, and vice versa, respectively. This software life cycle is designed to specifically support the development of ensembles characterized by their complexity and self-* properties, such as self-awareness, self-expression and self-adaptation.

The *offline* activities are grouped into requirements engineering, modeling and programming, and verification and validation phases. In addition to model functional and non-functional requirements as in traditional requirements engineering, the focus is on modeling aspects of self-adaptation and self-awareness. These specific requirements need to be validated and verified as well.

The *online* activities comprise monitoring, awareness and self-adaptation. Monitoring consists on the observation of the environment and the behavior of the systems. Reasoning on the collected data is also a key aspect. Finally, adaptation is performed in order to change the system according to the knowledge acquired during monitoring and the reasoning performed by the awareness engine.

In this paper we describe how we have applied the EDLC on the e-Mobility case study. In the spirit of EDLC, we have employed several interrelated methods developed in ASCENS

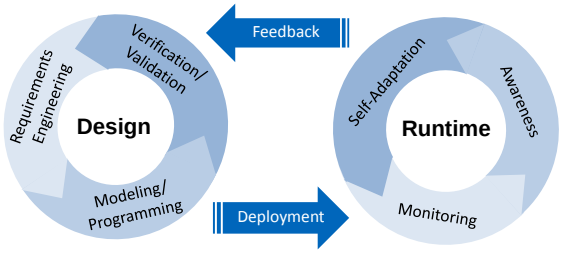


Figure 1. Ensemble Development Life Cycle (EDLC).

to address the application life cycle of the e-Mobility. In particular, we start with the specification of requirements and their reflection in the operation space of the system and system's self-awareness (described in Section IV). Following the requirements specification, we focus on the high-level architectural design, both in terms of adaptation patterns (Section V) and in goals decomposable into individual components and ensembles (Section VI). Next, emphasis is put on low-level design of component activities. For that, we employ the SCEL language, which is specifically intended for ensemble-based description of communication and coordination-oriented concerns (Section VII). Along with SCEL we also employ (Section VIII) the SCLP (soft-constraints logic programming), which provides a natural way of describing optimization related tasks, which are very frequent in self-adaptive systems. The final step is the implementation of components and the deployment of the system (Section IX), for which we use a dedicated ensemble-based component model and component runtime (called DEECo).

IV. Requirements Engineering with SOTA

Requirements engineering is of paramount importance to understand the adaptation needs of a system-to-be [2]. In the area of adaptive systems, and more in general of open-ended systems, the most appropriate approach is to adopt a goal-oriented requirements engineering one, and accordingly model requirements in terms of *goals* [3].

A goal represents a desirable state of the affairs that an entity, let it be a software component or an ensemble, aims to achieve. The idea of goal-oriented modeling of requirements naturally matches goal-oriented and intentional entities, such as organizations and multi-agent systems. However, self-adaptation too is naturally perceivable as an "intentional" quality. In fact, a self-adaptive component/ensemble should be engineered not simply to "achieve" a functionality or state of the affairs, but rather to "strive to achieve" such functionality, i.e., be able to take self-adaptive decisions and actions so as to preserve its capability of achieving despite unexpected contingencies and environmental changes.

Within the ASCENS development life cycle, SOTA [4] proposes itself as an extension of existing goal-oriented requirements engineering approaches that integrates elements of dynamical systems modeling, so as to account for the general needs of dynamic self-adaptive systems and components.

SOTA, which stems for "state of the affairs", models the entities of a self-adaptive system as if they were immersed in n-dimensional space $\mathbf{S}$, each of the n dimensions representing a specific aspect of the current situation of the entity/ensemble and of its operational environment. As the entity executes, its position in $\mathbf{S}$ changes either due to its specific actions or because of the dynamics of environment. Thus, system evolution can be seen as movements in $\mathbf{S}$.

The activity of requirements engineering for self-adaptive systems in SOTA requires identifying the dimensions of the SOTA space, which means modeling the relevant information that the different components and ensembles of a system have to collect to become aware of their location in such space. In e-mobility, the space $\mathbf{S}$ includes the spatial dimensions related to the street map, but also dimensions related to the current traffic conditions, the battery conditions, etc.

Once the SOTA space is defined, a goal in SOTA can be expressed in terms of a specific state of the affairs to aim for, that is, a specific point or a specific area in $\mathbf{S}$ which the component or ensemble should try to reach in its evolution, in spite of external contingencies that can move the trajectory farther from the goal. For instance, a goal for a vehicle could imply reaching a position in the SOTA space that, for the dimensions representing the spatial location, trivially represents the final destination and for the dimension representing the battery condition may represent a charging level ensuring safe return.

V. From SOTA to High-level Design with Adaptation Patterns

The SOTA modeling approach is very useful to understand and model the functional and adaptation requirements, and to check the correctness of such specifications (as described in [4]). However, when a designer considers the actual design of the system, it is important to identify which architectural schemes need to be chosen for the individual components and ensembles.

To this end, in previous work [5], we defined a taxonomy of architectural patterns for adaptive components and ensembles of components. This taxonomy has the twofold goal of enabling reuse of existing experiences and providing useful suggestions to a designer on selecting the most suitable patterns to support adaptability.

At the center of our taxonomy is the idea that self-adaptivity requires the presence (explicit or implicit) of a *feedback loop* or control loop. A feedback loop is the part of the system that allows for feedback and self-correction towards goal achievement, i.e., self-adjusting behavior in response to changes in the system. Feedback loops provide a generic mechanism for adaptation as they provide the means for inspecting and analyzing the system at the component or ensemble level and for reacting accordingly.

However, when it comes to choosing among a variety of possible architectural schemes that can be employed for feedback loops [5], it becomes clear that the specific characteristics of goals identified in the requirements engineering phase directly guides the choice of specific feedback loop patterns.

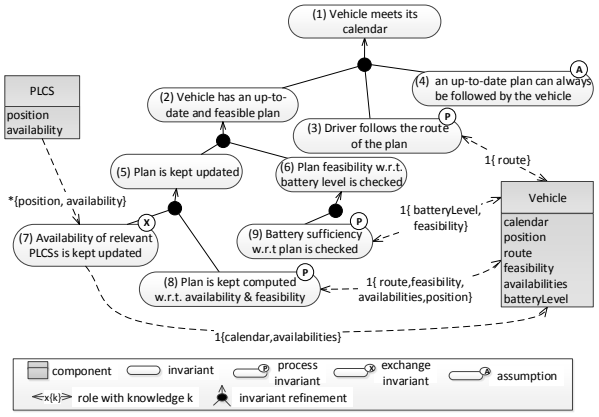


Figure 2. E-Mobility system level graph – IRM method.

In particular, the choice of a specific pattern depends on how (and to which extent) the components of the system have component-specific goals with very different characteristics, or rather they share the same ensemble-level goals or goals with very similar characteristics.

As an example, the *goal-oriented component* pattern considers autonomic components with internal control loops. Goal oriented behavior is explicit in the actions determined by the control loop, which can select actions to actively pursue goals in an adaptive way. A pattern of this kind, in which the feedback loop is embedded within the component, is suitable for those components that have very specific goals, that no other components share.

As another example, the *autonomic service component* pattern is characterized by an explicit external feedback loop. That is, the control loop is realized by "attaching" via appropriate interfaces an external controller that turns a simple service component into a component whose activities can be externally controlled to make the component itself goal-oriented and adaptive.

In a coordinated system for e-mobility, the e-vehicles of a car-sharing company may all share the same basic adaptation goals, thus making it suitable to model them as simple components all sharing the same class of external controller. Also, at the level of the fleet of e-vehicles, the presence of a single stakeholder makes it possible to exploit a pattern of an ensemble with a global control loop to orchestrate the overall behavior of the fleet.

VI. High-level Design – Architecture

In order to guide the design of an ensemble-based system from high-level strategic goals, requirements and patterns (described by SOTA) to their low-level realization in terms of system architecture (components and ensembles) we use the *Invariant Refinement Method* (IRM) [6].

The main idea of IRM is to capture the high-level system goals and requirements in terms of interaction *invariants*.

In compliance to SOTA's notion of "striving to achieve", invariants describe the desired state of the system-to-be at every time instant. In general, invariants are to be maintained by the coordination of the different system components. At the design stage, by *component* we refer to a participant or actor of the system-to-be. A special type of invariant, called *assumption*, describes a condition that is expected to hold about the environment; an assumption is not intended to be maintained explicitly by the system-to-be.

As a design decision, identified top-level invariants are decomposed into more concrete sub-invariants forming a decomposition graph (Figure 2). The decomposition is essentially a refinement, where the composition of the children exhibits all the behavior expected from the parent and (potentially) some more. By this decomposition, we strive to get to the level of abstraction where the (leaf) invariants represent detailed design of the particular system constituents – components, component processes, and ensembles. Two special types of invariants, namely the *process invariants* (denoted by "P") and *exchange invariants* (denoted by "X"), are used to model the low-level component computation (processes) and interaction (ensembles), respectively.

A possible system-level graph corresponding to the simplified e-Mobility scenario is depicted in Figure 2. In this case, the IRM design mainly captures the necessity to keep the vehicle's plan updated (invariant (5)) and to check whether the current plan remains feasible with respect to measured battery level (invariant (6)). The identified leaf invariants are easily mappable to component activities, which are further formally captured by SCEL or SCLP.

VII. Modeling Computational Activities: SCEL

To complement the high-level, architectural design, we have proposed specific linguistic and programming abstractions aiming at dealing with the challenges posed to language designers by massively distributed and highly dynamic systems. Our starting points have been the notions of *autonomic components* (ACs) and *autonomic-component ensembles* (ACEs) that are used to structure systems into independent and distributed building blocks that interact and adapt in different ways. Based on the notions of ACs and ACEs, we have introduced a number of specific abstractions and linguistic constructs that permit building up ACs, defining ACEs and programming their behaviors and interactions. The proposed abstractions are the basis of SCEL (Software Component Ensemble Language) [7], [8].

ACs are entities with dedicated knowledge units and resources that can cooperate while playing different roles. Each AC is equipped with an *interface*, consisting of a set of *attributes*, such as provided functionalities, spatial coordinates, group memberships, trust level, response time, etc. Attributes are used by the ACs to dynamically organize themselves into ACEs.

Indeed, one of the main novelties of SCEL is the way sets of partners are selected for interaction and thus how ensembles are formed. Individual ACs not only can single out communication partners by using their identities, but they can also select partners by exploiting the attributes in the interfaces of the individual ACs. Predicates over such attributes are used to specify the targets of communication actions, thus providing a sort of *attribute-based* communication. In this way, the formation rule of ACEs is endogenous to ACs: members of an ensemble are connected by the interdependency relations defined through predicates.

Starting from the IRM model presented in Figure 2, we can identify two kinds of SCEL components: PLCSs ad Vehicles. A PLCS identifies a *parking lot/charging station* and is characterized by a *position* and by its *availability*. These are the attributes that are exposed in the component interface and that respectively identify the location of the PLCS and the number of available slots in the area. As expected, Vehicle components identify cars involved in the scenario and will expose in their interface a set of attributes describing the state of the component (*position*, *batterylevel*,...). Attributes of both PLCSs and Vehicles are obtained as the projection on the interface of the local knowledge of each component.

The user associated to a vehicle is modeled by a process that, according to the local Vehicle interface, will interact with PLCSs in order to identify the next stop in the travel. This task is largely simplified thanks to the use of attribute based communication. Indeed, if *poi* is the next *point-of-interest* to visit in the travel, then the next PLCS to use can be identified by sending a *reservation request* to all the PLCSs components that are close to *poi* up-to a given *walking distance* and that can be reached with the current battery level.

However, when the battery level of a vehicle decreases under a given threshold, the actual behaviour can be adapted so to force the reservation of a PLCS that can be used to recharge the battery and then continue with planned trip.

VIII. Adaptation via Soft-Constraints Solving and Optimization

As a complement to SCEL specifically targeting intuitive specification of optimization problems that frequently appear in self-adaptive systems, we have used our approach on Soft Constraint Logic Programming.

Constraint logic programming (CLP) [9] extends logic programming (LP) by embedding constraints in it. However, only classical constraints can be handled. So, in [10], a further extension has been proposed to also handle soft constraints. This has led to a high-level and flexible declarative programming formalism, called *Soft CLP* (SCLP), allowing to easily model and solve real-life problems. Roughly speaking, SCLP programs are logic programs where logical constants and operations are replaced by those of the semiring (a structure representing the levels of satisfiability or the costs of a constraint). Consequently, assignments of variables to the items of the Herbrand universe yield the levels of satisfiability or the costs of the constraints.

We have applied the SCLP framework [11] to the e-Mobility travel optimization problem described in [12], by modelling in Ciao [13][2] two scenarios: the (i) trip; and (ii) journey optimization problems. A solution to (i) finds the best trip

[2]We would like to thank the Clip group for its technical support.

in terms of travel time and energy consumption, while (ii) determines the optimal sequence of trips, guaranteeing that the user reaches each appointment in time and that the state of charge of the electric vehicle never falls below a given threshold.

Besides optimizing trips and journeys of single users, that we can call local problems, the e-Mobility case study aims at solving global problems, involving large ensembles of vehicles. For such large problems, solution is often unfeasible, with both SCLP and more efficient tools. To tackle these, we propose a coordination of declarative and procedural knowledge: the global problem is decomposed into several local problems, which can be separately solved by the SCLP implementation (e.g. [13]), and whose parameters can be iteratively determined by a programmable coordination strategy. The latter guarantees a suboptimal, yet acceptable global solution.

Let us consider for example the parking optimization problem, which consists in finding the best parking lot for each vehicle of an ensemble in terms of three factors: the distance from the current location of the vehicle to the parking lot, the distance from the parking lot to the appointment location and the cost of the parking lot. Solving a global optimization procedure which assigns the best parking lot to each vehicle of the ensemble would be rather expensive, and also not flexible (replanning could require lots of time). So we propose to use SCLP to solve the local problems and some procedural language to programme the orchestrator. In this setting, SCLP is convenient since the orchestrator will be able to access much more easily the parameters of its fact/clause-based declarative implementation than an ordinary imperative module structure.

In particular, the orchestrator could be programmed using an extension of SCEL or simply Java. The orchestrator, after receiving the requests from the vehicles which want to park, asks the SCLP tool to solve the local optimization problems, determining the best parking lot for each vehicle. Then, it verifies if the local solutions all together form an admissible global solution, that is, if each parking lot is able to satisfy the requests of the vehicles planning to park in it. If it is so, the problem is solved, otherwise the orchestrator queries the declarative knowledge again, but now by increasing the costs of the parking lots which received too many requests. The procedure is repeated, with suitable variations, until a global solution is found. Notice that in this way the orchestrator has a hypothetical, transactional behavior, with the options of committing (a solution is found) or partially backtracking (on the parkings which are overfull).

IX. Implementation and deployment

Next steps in the EDLC, following the architectural design and detailed specification of component activities, is implementation and deployment. For these steps, we employ our DEECo (*Dependable Emergent Ensembles of Components*) component model [14] to provide us with the relevant software engineering abstractions that ease the programmers' tasks.

A component in DEECo, features execution model based on the MAPE-K autonomic loop [15]. In compliance with SCEL, it consists of (i) well-defined knowledge, being a set of knowledge items and (ii) processes that are executed periodically in a soft real-time manner. The component concept is complemented by the first-class ensemble concept. An ensemble stands as the only communication mechanism between DEECo components. It specifies a *membership* condition, according to which components are evaluated for participation. The evaluation is based on the components' knowledge (their *attributes* in SCEL). An ensemble also specifies what is to be communicated between the participants, that is, the appropriate knowledge exchange function. Similar to component processes, ensembles are invoked periodically in a soft real-time manner. (See Figure 3 for an excerpts of components and ensembles descriptions as found in the e-Mobility case study.)

```
1  component Vehicle features AvailabilityAggregator:
2    knowledge:
3      batteryLevel = 90%,
4      position = GPS(...),
5      calendar = [ POI(WORKPLACE, 9AM−1PM), POI(MALL, 2PM−3PM), ... ],
6      availabilities = [ ],
7      plan = { route = ROUTE(...), isFeasible = TRUE }
8    process computePlan:
9        in plan.isFeasible, in availabilities, in calendar, inout plan.route
10     function:
11       if (!plan.isFeasible) plan.route ← planner(calendar, availabilities)
12     scheduling: periodic( 5000ms )
13   ...
14 ensemble UpdateAvailabilityInformation:
15   coordinator: AvailabilityAggregator
16   member: AvailabilityAwareParkingLot
17   membership:
18     ∃ poi ∈ coordinator.calendar:
19 ‘     distance(member.position, poi.position) ≤ TRESHOLD &&
20       isAvailable(poi, member.availability)
21   knowledge exchange:
22     coordinator.availabilities ← { (m.id, m.availability) | m ∈ members }
23   scheduling: periodic( 2500ms )
```

Figure 3. Examples of identified DEECo components & ensembles.

In order to bring the above abstractions to practical use we have used jDEECo[3] – our reification of DEECo component model in Java. In jDEECo, components are intuitively represented as annotated Java classes, where component knowledge is mapped to class fields and processes to class methods. Similarly, appropriately annotated classes represent DEECo ensembles.

Once the necessary components and ensembles are coded, they are deployed in jDEECo runtime framework, which takes care of process and ensemble scheduling, as well as low-level distributed knowledge manipulation.

X. Evaluation and Related Work

Having described the application of EDLC to the e-Mobility case study, we relate it in this section to other approaches having the same aim and we describe benefits that we have observed in performing the case study. In particular, we structure this section along three main topics addressed in the case study, namely (i) requirements engineering and architectural design, (ii) modeling of activities, (iii) programming and deployment.

As to requirements engineering and architectural design in the area of autonomic systems, the most recognized approaches

[3] http://github.com/d3scomp/JDEECo

Table I. SUMMARY OF METHODS/TOOLS USED.

Requirements engineering:	SOTA
High-level design:	IRM
Process/activities modeling:	SCEL
Adaptation/optimization modeling:	SCLP
Implementation/runtime:	DEECo / jDEECo

are KAOS [16] and Tropos [17]. Similar to our approach, they fall into the category of goal modeling and elaboration, especially in the area of agent-based systems. In our experience, they provide a very solid ground for requirements engineering, but fall short to an extent when continuous control with self-adaptivity (as in the case of e-Mobility case study) is sought for. For this reason, we have employed SOTA and IRM, which are centered around the notion of continuously "striving to achieve" and thus make the reasoning about a guided evolution of a system easier.

As for the activity modeling, our approach builds on the body of work carried out in coordination languages (e.g., KLAIM [18]) and process algebras. However, it extends it by providing a tailored semantics to describe and reason about cooperating groups of components (i.e. ensembles). In the same vein, SCLP builds on the experience with constraint solving, but adds the option of soft-constraints and integration with SCEL. Indeed, in the e-Mobility case study, we found the interplay of SCEL with SCLP very useful for description of mutually related activities of interaction and coordination among vehicles combined with finding a tradeoff between local-global optimums (reflecting the need of harmonizing the selfish and cooperative concerns of vehices in the case-study).

Finally, at the programming and deployment stage, our approach has been backed up by DEECo component model and its Java-based reification jDEECo. In this respect, it is possible to find a plethora of component models and SOA-based approaches (e.g. SCA, Fractal, OSGi). However, these typically fall short in well-defined dynamicity (as captured by the ensembles) as well as in autonomicity and self-adaptation capabilities (as featured by the special design of components as distributed MAPE-K based entities). Similar problems apply even to the agent-based approaches with their Belief-Desire-Intention model (e.g., JADE). On the other hand, the explicit support of DEECo for ensembles and components – based on knowledge and cyclic activities – proved to make the transition from SOTA/IRM-based design (together with activities captured by SCEL/SCLP) to runtime very smooth.

XI. CONCLUSIONS AND FUTURE WORK

We have presented the EDLC framework for development of self-aware autonomic adaptive systems applied to the e-Mobility case study, a driving case-study in the ASCENS FP7 project. We have particularly shown the offline processes of EDLC, starting from requirements modeling and pattern identification with SOTA, to refinement of system invariants with IRM, ending in activity modeling with SCEL and SCLP formalisms. We have also outlined the programming and deployment phases using DEECo/jDEECo. The summary of methods and tools used is provided in Table I.

Due to space constraints and present work organization we have focused on requirements analysis, modeling and programming phases and deployment transaction of EDLC. The further phases (i) verification and validation of functional and non-functional properties at design and runtime and (ii) system evolution, where historical data monitored at runtime are used to improve the system design, are subject of the current and future research.

REFERENCES

[1] I. Sommerville, D. Cliff, R. Calinescu, J. Keen, T. Kelly, M. Kwiatkowska, J. Mcdermid, and R. Paige, "Large-scale complex IT systems," *Commun. ACM*, vol. 55, no. 7, pp. 71–77, Jul. 2012.

[2] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM TAAS*, vol. 4, no. 2, pp. 1–42, 2009.

[3] J. Mylopoulos, L. Chung, and E. S. K. Yu, "From Object-Oriented to Goal-Oriented Requirements Analysis," *Communications of the ACM*, vol. 42, no. 1, pp. 31–37, 1999.

[4] D. B. Abeywickrama and F. Zambonelli, "Model Checking Goal-Oriented Requirements for Self-Adaptive Systems," in *Proc. of ECBS*. IEEE, Apr. 2012, pp. 33–42.

[5] G. Cabri, M. Puviani, and F. Zambonelli, "Towards a Taxonomy of Adaptive Agent-based Collaboration Patterns for Autonomic Service Ensembles," in *Proc. of CTS*. IEEE, May 2011, pp. 508–515.

[6] 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 '13*. Vancouver, Canada: ACM, 2013.

[7] R. De Nicola, M. Loreti, R. Pugliese, and F. Tiezzi, "SCEL: a Language for Autonomic Computing," IMT Lucca, Tech. Rep., January 2013. [Online]. Available: http://rap.dsi.unifi.it/scel/pdf/SCEL-TR.pdf

[8] R. De Nicola, G. L. Ferrari, M. Loreti, and R. Pugliese, "A Language-Based Approach to Autonomic Computing," in *Revised Selected Papers of FMCO*. Springer, 2011, pp. 25–48.

[9] J. Jaffar and J. L. Lassez, "Constraint Logic Programming," in *POPL*. ACM Press, 1987, pp. 111–119.

[10] S. Bistarelli, U. Montanari, and F. Rossi, "Semiring-Based Constraint Logic Programming: Syntax and Semantics," *ACM TOPLAS*, vol. 23, no. 1, pp. 1–29, 2001.

[11] G. V. Monreale, U. Montanari, and N. Hoch, "Soft Constraint Logic Programming for Electric Vehicle Travel Optimization," *CoRR*, vol. abs/1212.2056, 2012.

[12] N. Hoch, K. Zemmer, B. Werther, and R. Y. Siegwarty, "Electric Vehicle Travel Optimization - Customer Satisfaction Despite Resource Constraints," in *Proc. of IEEE IVS*. IEEE, 2012.

[13] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla, "The Ciao Prolog System. Reference manual," School of Computer Science, Technical University of Madrid (UPM), Tech. Rep. CLIP3/97.1, 1997.

[14] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil, "DEECo – an Ensemble-Based Component System," in *Proc. of CBSE '13*. Vancouver, Canada: ACM, 2013.

[15] J. Kephart and D. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[16] A. V. Lamsweerde, "Requirements Engineering: from Craft to Discipline," in *SIGSOFT '08/FSE-16*. ACM, 2008, pp. 238–249.

[17] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos, "Tropos: An Agent-Oriented Software Development Methodology," *Autonomous Agents and Multi-Agent Systems*, vol. 8, no. 3, pp. 203–236, May 2004.

[18] R. De Nicola, G. Ferrari, and R. Pugliese, "KLAIM: A Kernel Language for Agents Interaction and Mobility," *Software Engineering, IEEE Transactions on*, vol. 24, no. 5, pp. 315–330, 1998.

Chapter 5

Evaluation Strategy

This chapter describes the strategy used throughout the research leading to this thesis in order to evaluate the proposed ideas and methods, in particular the Invariant Refinement Method (IRM), and its extension to self-adaptivity (IRM-SA). Where appropriate, we point to specific papers from Chapter 4 for detailed explanation and results.

The evaluation strategy is grounded in three complementary pillars (Figure 7), representing the three directions that were followed for evaluation:

1. Through case studies, by modeling non-toy examples of real-world applications stemming from industrial and project collaboration.

2. Through implementation of tools for graphical design, model transformations and code generation, and experimentation with self-adaptation with jDEECo.

3. Through an empirical study evaluating the feasibility and effectiveness of IRM-SA.

Each of the pillars 1 to 3 is further detailed in Sections 5.1 to 5.3, respectively.

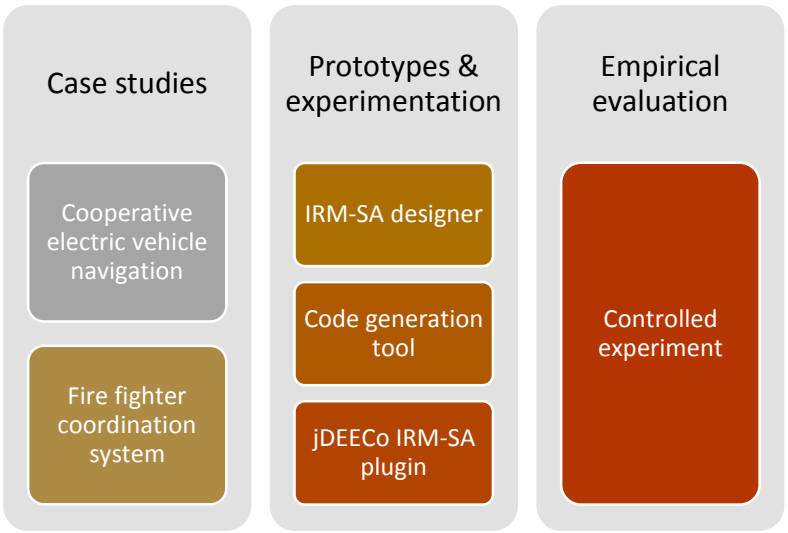


Figure 7. The three pillars of the employed evaluation strategy.

5.1 Evaluation through Case Studies

In order to evaluate the appropriateness of modeling siCPS with IRM and IRM-SA, we applied the methods in two real-life case studies.

The first case study concerns the design and implementation of a system for cooperative navigation of electric vehicles in a city – already overviewed in Section 1.1.1. This case study stemmed from one of the ASCENS project case studies [SRA+11]. It was further elaborated in a bilateral project between the Department of Distributed and Dependable Systems of Charles University and Volkswagen AG (only partial results of this project have been published, as the complete results fall under a non-disclosure agreement). The cooperative electric vehicle navigation case study has been an important driver of the research leading to this thesis; it was employed as running example in a set of papers including the papers presented in Sections 4.2, 4.3, 4.4, and 4.7.

The second case study concerns the design and implementation of an emergency coordination system for fire fighters. The case study was developed within the project DAUM, which serves as an exemplar for the evaluation of real-life real-scale distributed systems, developed in cooperation with professional fire fighters [4]. It has been employed as running example for the technical report included in Section 4.6. In the work reported in the same paper, an extended version of the running example was used in the experimental evaluation of IRM-SA-based jDEECo self-adaptation (Section 5.2.2.1).

5.2 Evaluation through Prototypes and Experimentation

To allow for experimentation with self-adaptation based on IRM-SA, tool support allowing the definition of IRM-SA models, structural checks, and code generation, along with an implementation of IRM-SA-based self-adaptation as a plugin to the jDEECo platform, was provided. These two (publicly available) projects are detailed in this section.

5.2.1 IRM-SA Design and Code Generation Tool

The IRM-SA design and code generation tool allows a designer to create an IRM-SA model using a graphical editor [14]. The designer can specify the (non-leaf) invariants, process and ensemble invariants, assumptions, and also the components and the component knowledge of the system-to-be. For each of these entities, different attributes can be specified. For example, for a process invariant, the signature and period of the process operationalizing it can be specified as an attribute. Invariants can be associated with each other through AND- and OR-decompositions; they can also be associated with components via contribution links. Validation via simple structural checks is also provided. For example, an IRM-SA model can be checked on whether each leaf invariant is associated with a component. Figure 8 depicts a snapshot of the IRM-SA graphical editor.

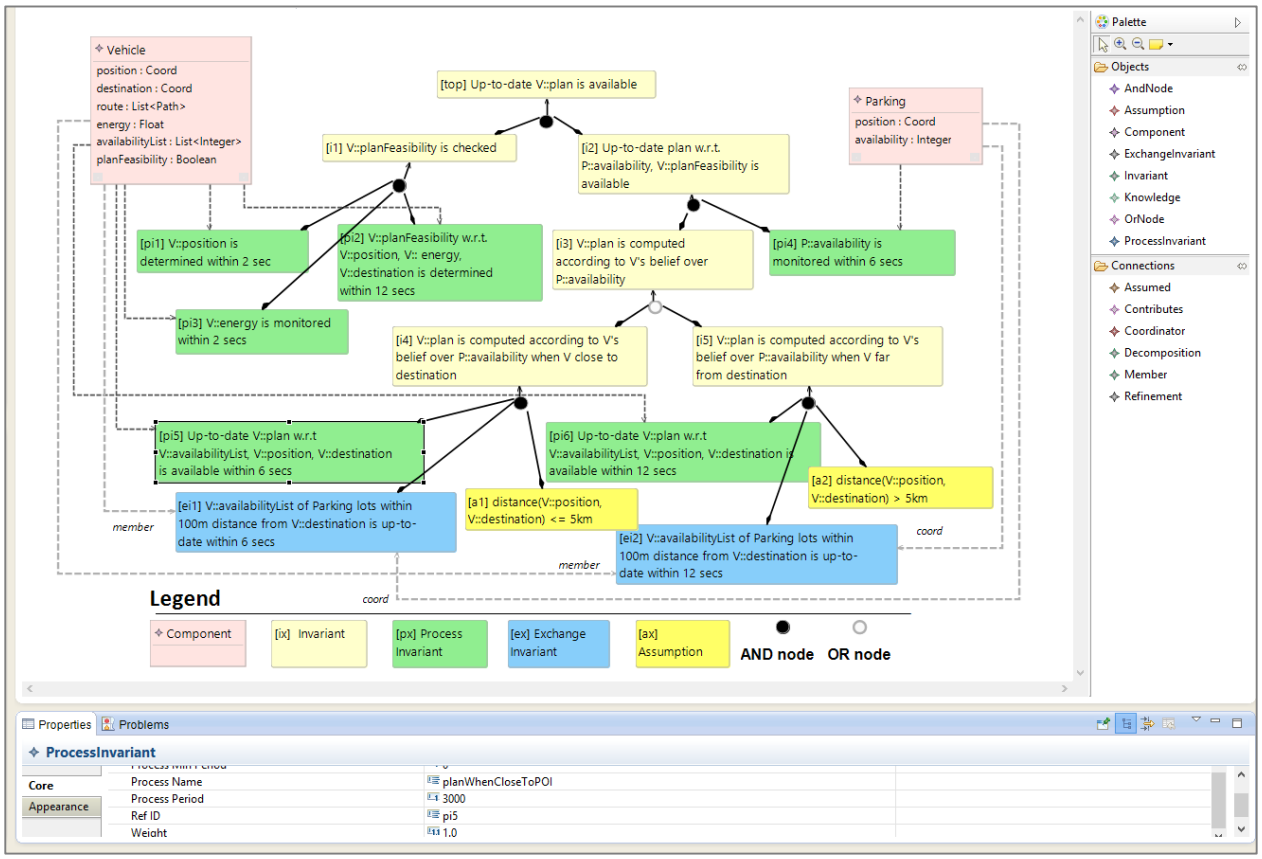


Figure 8. A snapshot of the IRM-SA graphical editor with a preliminary IRM-SA model of an example from the electric vehicle mobility case study.

Once an IRM-SA model is created and validated, it can be used for code generation. Since our target implementation and deployment platform was jDEECo, the code generation utility emits properly annotated Java code of jDEECo components and ensembles. The code generation also creates traceability links from the Java code to the IRM-SA models in the form of additional (IRM-SA-specific) jDEECo annotations. Technically, the generation is performed in two steps: first, the IRM-SA model is transformed into a DEECo model (model-to-model transformation), and then the DEECo model is used to generate the relevant code snippets (model-to-text transformation).

The tool has been implemented using Eclipse Modeling Framework (EMF) technologies, specifically the Epsilon tool suite [7]. All the models created or generated by the tool comply with the Ecore meta-model [6]. The implementation of the graphical editor is based on Eugenia [7], a front-end tool for Graphical Modeling Framework (GMF) [8], while the model-to-model and model-to-text transformations are based on Epsilon Transformation Language (ETL) [KRGDP14] and Epsilon Generation Language (EGL) [KRGDP14], respectively.

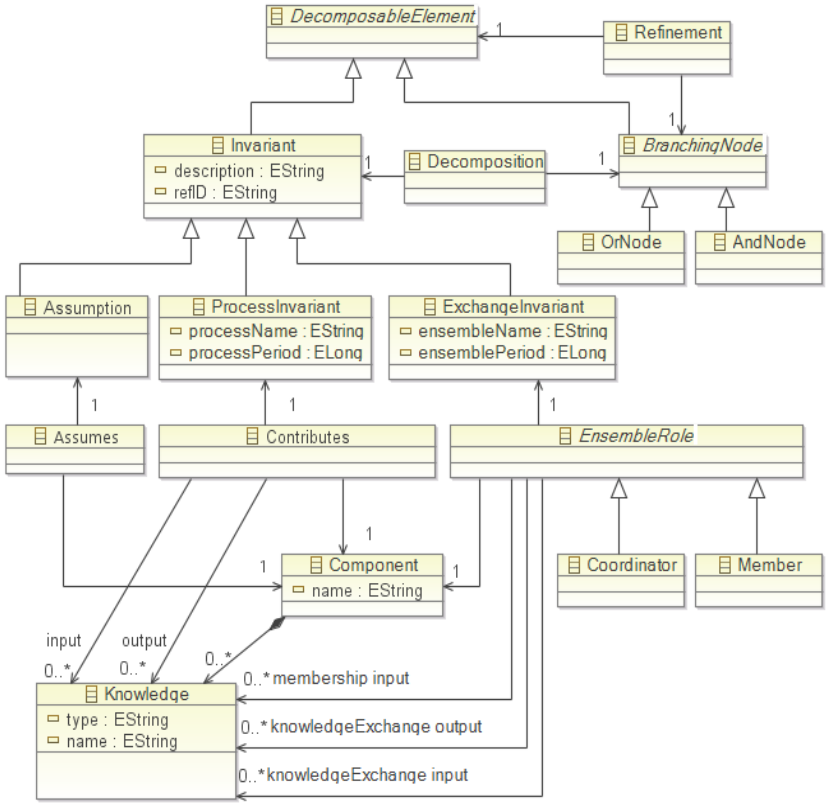


Figure 9. The IRM-SA meta-model employed in the IRM-SA design and code generation tool.

Among other uses, the IRM-SA design and code generation tool was used to create the IRM-SA model and generate the code skeletons for the non-trivial evaluation example of the fire fighter coordination system (comprising 4 components, 39 invariants, and 23 decompositions), reported in the technical report of Section 4.6.

The implementation of the tool helped in refining and validating the modeling concepts of IRM-SA. In particular, a first step towards creating the modeling tool was to formalize the IRM-SA concepts and their relations into an Ecore-compliant meta-model (Figure 9). This provided insight into the semantics of several syntactic constructs that were allowed for a more intuitive design. In particular, the semantics of (i) an invariant participating in the decomposition of more than one higher level invariants (*shared* invariant), and (ii) the chaining of AND- and OR-decompositions were clarified.

5.2.2 jDEECo IRM-SA Plugin

In order to enable the use of an IRM-SA model at runtime to provide runtime (architecture) reconfiguration via switching between alternative invariant decompositions, a

jDEECo plugin has been implemented [15]. The plugin builds on the requirements reflection idea and allows the application of reconfiguration actions so that the underlying DEECo system continuously meets its requirements, as prescribed in the IRM-SA model.

Technically, the plugin relies on EMF technology for the creation and dynamic update of the involved models. Internally, the plugin uses a Boolean Satisfiability Solver (Sat4j [LBP10][29]) to find an applicable configuration, according to the current state of the system as reflected at the model level. To propagate the changes done at the model level down to the running system, EMF listeners have been used. The IRM-SA-based self-adaptation for jDEECo was first introduced in the paper of Section 4.5, and further elaborated in the technical report of Section 4.6.

5.2.2.1 Experimentation in Decentralized Settings

Implementing the jDEECo IRM-SA plugin allowed the experimentation with self-adaptation in real-world settings comprising decentralized deployment and realistic communication delays. For this, the jDEECo platform (and its network plugins) was used as a test bed. Self-adaptation was performed *locally* within each DEECo deployment node, instead of having a global "oracle" (e.g. a process of a dedicated component that gathers the knowledge of every other component, plans, and disseminates the self-adaptation actions back to each component). Thus, each node was running its own reasoning cycle based on its local view of the world (its *belief*). The objective was to evaluate and quantify the effects of (i) outdated belief (because of delays in knowledge transmission over unreliable ad-hoc networks) in self-adaptation, and (ii) local self-adaptation actions (which can be conflicting) in system-level performance. The experiments' settings and results are reported in the technical report of Section 4.6.

5.3 Empirical Evaluation

A major claim of this thesis is that IRM-SA (and thus also IRM) helps in the design of siCPS that are architected based on the ensemble paradigm. To provide evidence for this claim, an empirical study was designed, piloted, and conducted. The study took the form of a controlled experiment with students. Regarding the object(s) of study, purpose, quality focus, perspective, and context (as prescribed in a standard goal template for empirical studies [BR88, WRH+12]), the scope of the experiment was to:

> Analyze *IRM-SA*
> for the purpose of *evaluation*
> with respect to *feasibility and effectiveness*
> from the point of view of *the researcher*
> in the context of *M.Sc. and Ph.D. students producing DEECo artifacts*.

In the experiment, the focus was on the design process that needs to be followed to create a DEECo architecture, i.e. on how to specify the appropriate DEECo components (together with their knowledge and processes) and DEECo ensembles, from an initial set of requirements (given as user stories). The main hypothesis was that using IRM-SA in

this process would increase the accuracy of the final architecture artifacts. To obtain a baseline, participants of a control group followed their intuition to translate the user stories directly to a DEECo architecture. On the contrary, participants that used IRM-SA (treatment group) first created an IRM-SA model of the system-to-be, by specifying the relevant components, invariants and associations between them, and then used the model to specify a DEECo architecture.

The experiment planning, results, and threats to validity are reported in the technical report of Section 4.6. The material handed to the participants of the control and treatment groups during the experiment (including the background material, task description, and questionnaires) are available online [13].

Overall, the results showed that (i) there is a need for guidance in the design of DEECo-based systems from initial requirements, as the ensemble-based modeling paradigm is rather new, and (ii) IRM-SA indeed helps, as it systematizes the design process and drives the focus on the timing requirements that the underlying processes need to meet.

Chapter 6

Conclusion & Open Challenges

This thesis has introduced and elaborated the *Invariant Refinement Method* (IRM), a method for the design of *software-intensive Cyber-Physical Systems* (siCPS) based on the *ensemble* paradigm. The grounding idea of IRM is to capture system requirements in terms of invariants, i.e. obligations that need to be continuously met, and iteratively decompose them until the level where they can be mapped one-to-one to system computation and communication activities. This thesis has also introduced and elaborated an extension of *IRM for self-adaptation* (IRM-SA). IRM-SA provides dependability in the form of (i) design patterns guiding the correct decomposition of higher-level invariants to lower-level ones, (ii) traceability between system requirements and architecture configurations, and (iii) mechanisms to deal with operational uncertainty. It provides self-adaptivity in the form of switching between configurations at runtime (architecture reconfiguration) to address specific situations that the system may reside in. IRM has been integrated into the *ensemble development life cycle*, a holistic process for building autonomic systems. The joint product of the above results leads us to the conclusion that the first research goal (**G1)**, as outlined in Section 1.3, has been achieved.

In order to evaluate the proposed methods and models, this thesis provided a mapping of IRM-SA concepts to concepts of *DEECo component model*, a component system following the ensemble paradigm. This was supported by a graphical editor and model-to-model and model-to-text transformations, which allow for tool-supported model-driven development of siCPS. Additionally, an implementation of self-adaptation based on IRM-SA was provided as a plugin to jDEECo platform, a Java reification of DEECo. This allows for quick experimentations with self-adaptation based on IRM-SA, so that problems can be identified and corrected early on in the development life cycle. The above results provide sufficient evidence for concluding that the second research goal (**G2)** has also been achieved.

Since the design and development of siCPS in a model-driven fashion is an overarching topic with long history (e.g. in the areas of embedded and real-time systems and multi-agent systems), but also pressing challenges for future research, this thesis does not attempt to address every aspect of the challenges related to model-driven development of siCPS. Rather, it aims to provide a pragmatic yet scientifically grounded baseline for further research in the area.

To conclude the thesis, in the remainder of this chapter, the author's subjective view on the important and exciting research challenges that remain open in the area of model-driven development of siCPS are presented.

Human in the loop. One of the disputes within the self-adaptive systems community is on the role of humans in the autonomic loop (and specifically with reference to the MAPE-K model for autonomic systems). This is particularly important for self-adaptive cyber-physical systems, as they are increasingly becoming sociotechnical systems with stringent requirements on their safe operation and large influence in social life [Eve14]. Challenging questions in this context include "To which extend should siCPS perform their actions without human supervision?" and "How to design and develop siCPS so that human users can intervene only *when* and *where* needed?"

In order to involve human users in the operation of siCPS, a crucial step is to understand what a system is doing (e.g. what adaptation action is undertakes) and why it is doing it, a property sometimes called *self-awareness* [HS06] or *self-explanation* [SBW+10]. This idea has been pointed out in research around *requirements reflection* [BWS+10].

A promising direction is to establish a feedback between runtime and design. This has been identified as an essential step in the "two-wheel" model of the ensemble development life cycle [HKP+15]. A particular challenge is to provide the user with *as much information as needed* about the running system *and not more*, in order to be able to process the information needed for evolving the system. In this direction, approaches for analysis and visualization of large amounts of data (e.g. mega-modeling of big data analytics [CVPT12]) seem promising.

Another direction is to investigate sophisticated methods for decision making, which do not involve human users directly, but take into account user preferences and priorities – so-called *user-centric adaptation* [GGPTRC14]. In this context, research on integrating non-functional requirements, whose satisfaction criteria are elicited by stakeholders [ZSL14], in the runtime decision-making process (the Analyze and Plan phases of the MAPE-K loop) is highly relevant.

Ideally, self-adaptive siCPS should decide by themselves when to involve humans in their operation and in which way. These decisions invariably need to take into account the unpredictability and spontaneity of human agents. One promising direction is to formalize human behaviors and the factors that affect them (e.g. as stochastic multi-player games [CMG15]) and use this formalization in the analysis and design of human-system-environment interactions.

Taming uncertainty. Uncertainty appears lately as a recurrent topic in the software engineering community, as there is increasing awareness over uncertainty aspects that creep in at different phases of the software development life cycle, from requirements elicitation to operation [Gar10, LSB14, TMC+13, WSB11]. Indeed, it seems that uncertainty in user needs, assumptions about the environment, behavior of third-party components, etc. is the primary driver for making software systems self-adaptive. While a driver for self-adaptivity, uncertainty (or rather ignoring uncertainty) may lead to a severe decrease in safety and dependability.

In siCPS, uncertainty in the norm, not the exception: siCPS usually operate in environments difficult to anticipate and model a priori; they possess multiple loci of control resulting in complex and often emergent behaviors; they continuously interact with unpredictable users and imprecise and unreliable sensors and actuators.

While research in requirements engineering methods to deal with uncertainty (e.g. as in [LSB14, WSB+10]) is important, a largely open research challenge in siCPS – elaborated also in Section 1.2 – is related to finding ways to fight *operational* or *external* uncertainty, i.e. the uncertainty that arises from the environment in which the system in deployed [EKM11]. In IRM, we have provided a mechanism to deal with external uncertainty that relies on reasoning on the inaccuracy of components' belief [GBH+15b].

A direct result of external uncertainty is that it makes it hard to anticipate at design time all possible situations that a system may reside in and to provide corresponding adaptation actions. In such cases, a promising idea is to endow a system with some kind of meta-adaptation mechanism that allows the adaptation of the adaptation logic itself (e.g. via employing models@runtime [PMC+12]) to deal with unforeseen situations. In [GBH+15a], we have discussed a possible realization of this idea in the domain of siCPS.

Alignment of software engineering with other disciplines. One of the grand challenges in software engineering of siCPS is to devise ways to integrate software engineering principles and practices with disciplines such as mechanical and electrical engineering, control engineering, and physics [DLV12, KK12]. This alignment is challenging, as different disciplines adopt different views over the siCPS world, and base their models and analysis and design methods in different sets of assumptions. It is also essential, as siCPS is by its own nature a complex multi-disciplinary domain.

One of the relevant questions is on which basis to attempt such an alignment. A promising direction is to use *software architecture models* as the common vocabulary across disciplines and as the vehicle to map different views into a representation that is commonly understood and used by software engineers [Gar15]. In this respect, software architectures become "richer", as they integrate information not only about the structure of a siCPS, but also about other aspects such as performance or even physical constraints and laws. In [AABG+14b], we have explored this idea by integrating laws about the physical evolution of data in a distributed system (e.g. position, velocity, acceleration of vehicles modeled as autonomous components), in the form of ordinary differential equations, into the architecture description of a siCPS.

In the development of siCPS with self-adaptive capabilities, there also seems to be a large potential in integrating well-studied and formally proven techniques from control engineering [FGLM11, FMA+15]. Control engineering also gains benefit from a new, exciting, and highly challenging field of application.

The software engineering community is only beginning to investigate the alignment (or, rather, the lack thereof) between software engineering and other disciplines in the siCPS domain and beyond. More research is needed in understanding, quantifying and bridging this gap in order to unlock the potential for game-changing applications of software-intensive cyber-physical systems.

References

[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. Author's copy 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*, pages 87–90. IEEE, April 2014.

[AAGGH+14] R. Al Ali, I. Gerostathopoulos, I. Gonzalez-Herrera, A. Juan-Verdejo, M. Kit, and B. Surajbali. An Architecture-Based Approach for Compute-Intensive Pervasive Systems in Dynamic Environments. In *HotTopiCS '14: Proceedings of the 2nd International Workshop on Hot Topics in Cloud service Scalability*. ACM, March 2014. Article no. 3.

[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 IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 48–53. IEEE, 2012.

[AGH+10] D. Amyot, S. Ghanavati, J. Horkoff, G. Mussbacher, L. Peyton, and E. Yu. Evaluating Goal Models Within the Goal-oriented Requirement Language. *Int. J. Intell. Syst.*, 25(8):841–877, August 2010.

[AGH+11] P. Avgeriou, J. Grundy, J. G. Hall, P. Lago, and I. Mistrik. *Relating Software Requirements with Architecture*. Springer, 2011.

[AK09] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[AN05] J. Arlow and I. Neustadt. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison-Wesley, 2005.

[Ant96] A. Anton. Goal-based requirements analysis. In *ICSE '96: Proceedings of the Second International Conference on Requirements Engineering*, pages 136–144. IEEE, April 1996.

[BBE+07] C. Barreto, V. Bullard, T. Erl, J. Evdemon, D. Jordan, K. Kand, S. Moser, R. Stout, R. Ten-hove, I. Trickovic, D. V. D. Rijn, and A. Yiu. Web Services Business Process Execution Language Version 2.0, 2007. http://docs.oasis-open.org/wsbpel/2.0/-OS/wsbpel-v2.0-OS.html.

[BBF09] G. Blair, N. Bencomo, and R. France. Models@ run.time. *Computer*, 42(10):22–27, October 2009.

[BBG+15] L. Bulej, T. Bures, I. Gerostathopoulos, V. Horky, J. Keznikl, L. Marek, M. Tschaikowski, M. Tribastone, and P. Tuma. Supporting Performance Awareness in Autonomous Ensembles. In M. Wirsing, M. Hölzl, N. Koch, and P. Mayer, editors, *Software Engineering for Collective Autonomic Systems*, volume 8998 of *Lecture Notes in Computer Science*, pages 291–322. Springer International Publishing, 2015.

[BCL+06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. The Fractal component model and its support in Java. *Software: Practice & Experience*, 36(11-12):1257–1284, 2006.

[BDNG+13] T. Bures, R. De Nicola, I. Gerostathopoulos, N. Hoch, M. Kit, N. Koch, G. Valentina Monreale, U. Montanari, R. Pugliese, N. Serbedzija, M. Wirsing, and F. Zambonelli. A Life Cycle for the Development of Autonomic Systems: The e-Mobility Showcase. In *SASOW '13: Proceedings of the 7th IEEE International Conference on Self-Adaptation and Self-Organizing Systems Workshops*, pages 71–76. IEEE, September 2013.

[Bec99] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, Reading, MA, October 1999.

[BFCA14] N. Bencomo, R. France, B. H. C. Cheng, and U. Aßmann, editors. *Models@run.time*, volume 8378 of *Lecture Notes in Computer Science*. Springer International Publishing, Cham, 2014.

[BGAA14] T. Bures, I. Gerostathopoulos, and R. Al Ali. DEECo: Software Engineering for Smart CPS. *ERCIM News*, April 2014. Published online: http://ercim-news.ercim.eu/en97/special/deeco-software-engineering-for-smart-cps.

[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*, pages 250–266. Springer, August 2014.

[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, Department of Distributed and Dependable Systems, April 2014. Available online: http://-d3s.mff.cuni.cz/publications/.

[BGH+15] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil. The Invariant Refinement Method. In M. Wirsing, M. Hölzl, N. Koch, and P. Mayer, editors, *Software Engineering for Collective Autonomic Systems*, volume 8998 of *Lecture Notes in Computer Science*, pages 405–428. Springer International Publishing, 2015.

[BGK+15] T. Bures, I. Gerostathopoulos, J. Keznikl, F. Plasil, and P. Tuma. Formalization of Invariant Patterns for the Invariant Refinement Method. In R. De Nicola and R. Hennicker, editors, *Software, Services and Systems*, volume 8950 of *Lecture Notes in Computer Science*, pages 602–208. Springer International Publishing, 2015.

[BGPP03] C. Bernon, M.-P. Gleizes, S. Peyruqueou, and G. Picard. ADELFE: A Methodology for Adaptive Multi-agent Systems Engineering. In P. Petta, R. Tolksdorf, and F. Zambonelli, editors, *Engineering Societies in the Agents World III*, volume 2577 of *Lecture Notes in Computer Science*, pages 156–169. Springer Berlin Heidelberg, 2003.

[BH06] R. H. Bordini and J. F. Hübner. BDI Agent Programming in AgentSpeak Using Jason. In F. Toni and P. Torroni, editors, *Computational Logic in Multi-Agent Systems*, volume 3900 of *Lecture Notes in Computer Science*, pages 143–164. Springer Berlin Heidelberg, 2006.

[BHP06] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48. IEEE, August 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.

[BKW03] D. Berry, R. Kazman, and R. Wieringa. *STRAW'03: Proceedings of the Second International SofTware Requirements to Architectures Workshop*. 2003.

[BMO01] B. Bauer, J. P. Müller, and J. Odell. Agent UML: A Formalism for Specifying Multiagent Software Systems. In P. Ciancarini and M. J. Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *Lecture Notes in Computer Science*, pages 91–103. Springer Berlin Heidelberg, 2001.

[Boe00] B. Boehm. Spiral Development: Experience, Principles, and Refinements. Special Report CMU/SEI-2000-SR-008, Carnegie Mellon University, July 2000. Available online: http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5053.

[BP10a] L. Baresi and L. Pasquale. Adaptive Goals for Self-Adaptive Service Compositions. In *ICWS '10: Proceedings of the 2010 IEEE International Conference on Web Services*, pages 353–360. IEEE, July 2010.

[BP10b] L. Baresi and L. Pasquale. Live Goals for Adaptive Service Compositions. In *SEAMS '10: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 114–123. ACM, 2010.

[BP11] L. Baresi and L. Pasquale. Adaptation Goals for Adaptive Service-Oriented Architectures. In P. Avgeriou, J. Grundy, J. G. Hall, P. Lago, and I. Mistrík, editors, *Relating Software Requirements and Architectures*, pages 161–181. Springer, 2011.

[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, May 2004.

[BPS10] L. Baresi, L. Pasquale, and P. Spoletini. Fuzzy Goals for Requirements-driven Adaptation. In *RE '10: Proceedings of the 18th IEEE International Requirements Engineering Conference*, pages 125–134. IEEE, September 2010.

[BR88] V. Basili and H. Rombach. The TAME project: Towards Improvement-Oriented Software Environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, June 1988.

[Bra99] M. E. Bratman. *Intention, Plans, and Practical Reason*. Center for the Study of Language and Information, Stanford, California, USA, March 1999.

[BWS+10] N. Bencomo, J. Whittle, P. Sawyer, A. Finkelstein, and E. Letier. Requirements Reflection: Requirements as Runtime Entities. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 199–202. IEEE, 2010.

[CKM01] J. Castro, M. Kolp, and J. Mylopoulos. A Requirements-Driven Development Methodology. In *CAiSE '01: Proceedings of the 13th International Conference on Advanced Information Systems Engineering*, pages 108–123. Springer Berlin Heidelberg, 2001.

[CKM02] J. Castro, M. Kolp, and J. Mylopoulos. Towards requirements-driven information systems engineering: the Tropos project. *Information Systems*, 27(6):365–389, 2002.

[CL02] I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[CLG+09] B. Cheng, R. d. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. D. M. Serugendo, S. Dustdar, A. Finkelstein, C. Cacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Muller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems*, pages 1–26. Springer Berlin Heidelberg, 2009.

[CMG15] J. Camara, G. A. Moreno, and D. Garlan. Reasoning about Human Participation in Self-Adaptive Systems. In *SEAMS '15: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2015. To appear.

[CNYM99] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Springer, 1999.

[Coh09] M. Cohn. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional, Upper Saddle River, NJ, November 2009.

[CSVC11] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. Chaudron. A Classification Framework for Software Component Models. *IEEE Transactions on Software Engineering*, 37(5):593–615, 2011.

[CVPT12] S. Ceri, E. D. Valle, D. Pedreschi, and R. Trasarti. Mega-modeling for Big Data Analytics. In P. Atzeni, D. Cheung, and S. Ram, editors, *Conceptual Modeling*, volume 7532 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2012.

[DBP94] E. Dubois, P. D. Bois, and M. Petit. ALBERT: An Agent-Oriented Language for Building and Eliciting Requirements for Real-Time Systems. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, pages 713 – 722. IEEE, 1994.

[DeL14] S. A. DeLoach. O-MaSE: An Extensible Methodology for Multi-agent Systems. In O. Shehory and A. Sturm, editors, *Agent-Oriented Software Engineering*, pages 173–191. Springer Berlin Heidelberg, 2014.

[DLV12] P. Derler, E. a. Lee, and a. S. Vincentelli. Modeling Cyber-Physical Systems. *Proceedings of the IEEE*, 100(1):13–28, January 2012.

[DNFLP13] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese. A Language-based Approach to Autonomic Computing. *Formal Methods for Components and Objects*, 7542:25–48, 2013.

[DVL96] R. Darimont and A. Van Lamsweerde. Formal Refinement Patterns for Goal-Driven Requirements Elaboration. In *FSE'96: 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 179–190. ACM, 1996.

[DVLF93] A. Dardenne, A. Van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1):3–50, April 1993.

[DW04] K. H. Dam and M. Winikoff. Comparing Agent-Oriented Methodologies. In P. Giorgini, B. Henderson-Sellers, and M. Winikoff, editors, *Agent-Oriented Information Systems*, volume 3030 of *Lecture Notes in Computer Science*, pages 78–93. Springer Berlin Heidelberg, 2004.

[DWS01] S. A. Deloach, M. F. Wood, and C. H. Sparkman. Multiagent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(03):231–258, June 2001.

[EKM11] N. Esfahani, E. Kouroshfar, and S. Malek. Taming Uncertainty in Self-Adaptive Software. In *SIGSOFT/FSE '11: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 234–244. ACM, 2011.

[Eve14] C. Evers. *The Human in the Loop: User Participation in Self-Adaptive Software*. PhD thesis, Faculty of Electrical Engineering and Computer Science University of Kassel, August 2014.

[FBP+12] F. Fouquet, O. Barais, N. Plouzeau, J.-m. Jezequel, B. Morin, and F. Fleurey. A Dynamic Component Model for Cyber Physical Systems. In *CBSE'12: Proceedings of International ACM SIGSOFT Symposium on Component Based Software Engineering*, pages 135–144. ACM, 2012.

[FGKM01] A. Fuxman, P. Giorgini, M. Kolp, and J. Mylopoulos. Information Systems as Social Structures. In *FOIS '01: Proceedings of the International Conference on Formal Ontology in Information Systems*, pages 10–21. ACM, 2001.

[FGLM11] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio. Self-Adaptive Software Meets Control Theory: A Preliminary Approach Supporting Reliability Requirements. In *ASE '11: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 283–292. IEEE, 2011.

[FLM+04] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. Specifying and Analyzing Early Requirements in Tropos. *Requirements Engineering*, 9(2):132–150, March 2004.

[FMA+15] A. Filieri, M. Maggio, K. Angelopoulos, N. D'Ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein, F. Krikava, S. Misailovic, A. V. Papadopoulos, S. Ray, A. M. Sharifloo, S. Shevtsov, M. Ujma, and T. Vogel. Software Engineering Meets Control Theory. In *SEAMS'15: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, May 2015. In press. Author's copy available online: http://d3s.mff.cuni.cz/publications/.

[Fow03] M. Fowler. *UML Distilled*. Addison-Wesley, 2003.

[FPMT01] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model Checking Early Requirements Specifications in Tropos. In *RE'01: Proceedings of the Fifth International Symposium on Requirements Engineering*, pages 174–181. IEEE, August 2001.

[Gar10] D. Garlan. Software Engineering in an Uncertain World. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, pages 125–128. ACM, 2010.

[Gar15] D. Garlan. Modeling Challenges for CPS Systems. In *Proceedings of the 1st International Workshop in Software Engineering for Smart Cyber-Physical Systems, Keynote Talk*. IEEE, 2015. In press.

[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, April 2013.

[GBH+15a] I. Gerostathopoulos, T. Bures, P. Hnetynka, A. Hujecek, F. Plasil, and D. Skoda. Meta-Adaptation Strategies for Adaptation in Cyber-Physical Systems. In *ECSA '15: Proceedings of the 9th European Conference on Software Architecture*. Springer, September 2015. To appear. Author's copy available online: http://-d3s.mff.cuni.cz/publications.

[GBH+15b] I. Gerostathopoulos, T. Bures, P. Hnetynka, J. Keznikl, M. Kit, F. Plasil, and N. Plouzeau. Self-Adaptation in Cyber-Physical Systems: from System Goals to Architecture Configurations. Technical Report D3S-TR-2015-02, Department of Distributed and Dependable Systems, April 2015. Available online: http://-d3s.mff.cuni.cz/publications/.

[Ger14] I. Gerostathopoulos. Model-Driven Design of Ensemble-Based Component Systems. In *MODELS '14: Proceedings of the ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems Poster Session and the ACM Student Research Competition*, volume 1258, pages 63–68. CEUR-WS.org, September 2014.

[GGK+15] S. Götz, I. Gerostathopoulos, F. Krikava, A. Shahzada, and R. Spalazzese. Adaptive Exchange of Distributed Partial Models@run.time for Highly Dynamic Systems. In *SEAMS'15: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, May 2015. In press. Authors' copy available online: http://d3s.mff.cuni.cz/publications/.

[GGPTRC14] J. García-Galán, L. Pasquale, P. Trinidad, and A. Ruiz-Cortés. User-Centric Adaptation of Multi-tenant Services: Preference-Based Analysis for Service Reconfiguration. In *SEAMS '14: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 65–74. ACM, 2014.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[GKB+14] I. Gerostathopoulos, J. Keznikl, T. Bures, M. Kit, and F. Plasil. Software Engineering for Software-Intensive Cyber-Physical Systems. In *INFORMATIK 2014: Proceedings of the 44th Annual Meeting of the German Informatics Society*, pages 1179–1190. Gesellschaft für Informatik, Bohn, Germany, September 2014.

[GKMP04] P. Giorgini, M. Kolp, J. Mylopoulos, and M. Pistore. The Tropos Methodology: An Overview. In *Methodologies and Software Engineering for Agent Systems*, pages 89–106. Kluwer Academic Publishers, 2004.

[Gom93] H. Gomaa. *Software Design Methods for Concurrent and Real-Time Systems*. Addison-Wesley Professional, Reading, Mass, August 1993.

[GS93] D. Garlan and M. Shaw. An Introduction to Software Architecture. In *Advances in Software Engineering and Knowledge Engineering*, pages 1–39. World Scientific, 1993.

[GS14] J. J. Gomez-Sanz. Ten Years of the INGENIAS Methodology. In O. Shehory and A. Sturm, editors, *Agent-Oriented Software Engineering*, pages 193–209. Springer Berlin Heidelberg, 2014.

[HBHM99] K. V. Hindriks, F. S. D. Boer, W. V. D. Hoek, and J.-J. C. Meyer. Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, November 1999.

[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.

[Her96] Herbert A. Simon. *The Sciences of the Artificial*. MIT Press, 3rd edition, 1996.

[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.

[HK14] R. Hennicker and A. Klarl. Foundations for Ensemble Modeling – The Helena Approach. In S. Iida, J. Meseguer, and K. Ogata, editors, *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.

[HKP+15] M. Hölzl, N. Koch, M. Puviani, M. Wirsing, and F. Zambonelli. The Ensemble Development Life Cycle and Best Practices for Collective Autonomic Systems. In M. Wirsing, M. Hölzl, N. Koch, and P. Mayer, editors, *Software Engineering for Collective Autonomic Systems*, volume 8998 of *Lecture Notes in Computer Science*, pages 325–354. Springer International Publishing, 2015.

[HPMS11] R. Hall, K. Pauls, S. McCulloch, and D. Savage. *OSGi in Action: Creating Modular Applications in Java*. Manning Publications, Stamford, CT, 2011.

[HPS12] M. Hinchey, S. Park, and K. Schmid. Building Dynamic Software Product Lines. *Computer*, 45(10):22–26, October 2012.

[HRW08a] M. Hölzl, A. Rauschmayer, and M. Wirsing. Software engineering for ensembles. In *Software-Intensive Systems and New Computing Paradigms*, pages 45–63. 2008.

[HRW08b] M. Hoelzl, 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*, pages 1–44. 2008.

[HS06] M. Hinchey and R. Sterritt. Self-managing software. *Computer*, 39(2):107–109, February 2006.

[IBM06] IBM. An architectural blueprint for autonomic computing. *IBM White Paper*, June 2006. Available online: http://www-03.ibm.com/autonomic/pdfs/AC Blueprint White Paper V7.pdf.

[IST11] P. Inverardi, R. Spalazzese, and M. Tivoli. Application-Layer Connector Synthesis. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 148–190. 2011.

[Jen00] N. R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2):277–296, March 2000.

[Kav02] E. Kavakli. Goal-Oriented Requirements Engineering: A Unifying Framework. *Requirements Engineering*, 6(4):237–251, January 2002.

[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.

[KBPK12] J. Keznikl, T. Bures, F. Plasil, and M. Kit. Towards Dependable Emergent Ensembles of Components: The DEECo Component Model. In *Proceedings of 2012 Joint Working Conference on Software Architecture (WICSA) & 6th European Conference on Software Architecture (ECSA)*, pages 249–252. IEEE, August 2012.

[KC01] J. Kramer and J. Castro. *STRAW'01: Proceedings of the First International SofTware Requirements to Architectures Workshop*. 2001.

[KC03] J. Kephart and D. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[KCH+90] K. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[KCH14] A. Klarl, L. Cichella, and R. Hennicker. From Helena Ensemble Specifications to Executable Code. In I. Lanese and E. Madelaine, editors, *Formal Aspects of Component Software*, volume 8997 of *Lecture Notes in Computer Science*, pages 183–190. Springer International Publishing, September 2014.

[KCM01] M. Kolp, J. Castro, and J. Mylopoulos. A Social Organization Perspective on Software Architectures. In *STRAW'01: Proceedings of the First International SofTware Requirements to Architectures Workshop*, 2001. Available online: http://-www.cin.ufpe.br/~straw01/.

[Kez14] J. Keznikl. *Dynamic Software Architectures for Resilient Distributed Systems*. PhD thesis, Charles University in Prague, Department of Distributed and Dependable Systems, 2014.

[KG97] D. Kinny and M. Georgeff. Modelling and Design of Multi-Agent Systems. In J. P. Müller, M. J. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III Agent Theories, Architectures, and Languages*, volume 1193 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 1997.

[KGB+15] M. Kit, I. Gerostathopoulos, T. Bures, P. Hnetynka, and F. Plasil. An Architecture Framework for Experimentations with Self-Adaptive Cyber-Physical Systems. In *SEAMS'15: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, May 2015. In press. Author's copy available online: http://d3s.mff.cuni.cz/publications/.

[KGM01] M. Kolp, P. Giorgini, and J. Mylopoulos. A Goal-Based Organizational Perspective on Multi-Agent Architectures. In *ATAL '01: Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages*, pages 128–140. Springer, 2001.

[KGM06] M. Kolp, P. Giorgini, and J. Mylopoulos. Multi-Agent Architectures as Organizational Structures. *Autonomous Agents and Multi-Agent Systems*, 13(1):3–25, February 2006.

[KH14] A. Klarl and R. Hennicker. Design and Implementation of Dynamically Evolving Ensembles with the Helena Framework. In *ASWEC'14: Proceedings of the 23rd Australian Software Engineering Conference*, pages 15–24. IEEE, April 2014.

[KK12] B. K.-d. Kim and P. R. Kumar. Cyber-Physical Systems: A Perspective at the Centennial. *Proceedings of the IEEE*, 100(Special Centennial):1287–1308, 2012.

[KMH14] A. Klarl, P. Mayer, and R. Hennicker. Helena@Work: Modeling the Science Cloud Platform. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 8802 of *Lecture Notes in Computer Science*, pages 99–116. Springer Berlin Heidelberg, October 2014.

[Koy92] R. Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. Springer Berlin Heidelberg, 1992.

[KRGDP14] D. Kolovos, L. Rose, A. García-Domínguez, and R. Paige. The Epsilon Book, April 2014. Available online: https://eclipse.org/epsilon/doc/book/.

[LBP10] D. Le Berre and A. Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[Lee08] E. A. Lee. Cyber Physical Systems: Design Challenges. In *ISORC'08: Proceedings of the 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing*, pages 363–369. IEEE, May 2008.

[LSB14] E. Letier, D. Stefan, and E. T. Barr. Uncertainty, Risk, and Information Value in Software Requirements and Architecture. In *ICSE '14: Proceedings of the 36th International Conference on Software Engineering*, pages 883–894. ACM, 2014.

[LVL02] E. Letier and A. Van Lamsweerde. Deriving Operational Software Specifications from System Goals. In *FSE'02: Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 119–119. ACM, 2002.

[LW07] K.-K. Lau and Z. Wang. Software Component Models. *IEEE Transactions on Software Engineering*, 33(10):709–724, October 2007.

[LYM03] L. Liu, E. Yu, and J. Mylopoulos. Security and Privacy Requirements Analysis within a Social Setting. In *RE'03: Proceedings of the 11th IEEE International Requirements Engineering Conference*, pages 151–161. IEEE, September 2003.

[Mal12] M. Malohlava. *Variability of Execution Environments for Component-based Systems*. PhD thesis, Department of Distributed and Dependable Systems, Charles University in Prague, 2012.

[MBJ+09] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models at Runtime to Support Dynamic Adaptation. *Computer*, 42(10):44–51, 2009.

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing Knowledge About Information Systems. *ACM Trans. Inf. Syst.*, 8(4):325–362, October 1990.

[MCN92] J. Mylopoulos, L. Chung, and B. Nixon. Representing and Using Nonfunctional Requirements: A Process-Oriented Approach. *IEEE Transactions on Software Engineering*, 18(6):483–497, June 1992.

[MCY99] J. Mylopoulos, L. Chung, and E. Yu. From Object-Oriented to Goal-Oriented Requirements Analysis. *Communications of the ACM*, 42(1):31–37, 1999.

[Mor11] M. Morandini. *Goal-Oriented Development of Self-Adaptive Systems*. PhD thesis, University of Trento, 2011.

[MP08] M. Morandini 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, May 2008.

[MPP08a] M. Morandini, L. Penserini, and A. Perini. Automated Mapping from Goal Models to Self-Adaptive Systems. In *ASE'08: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 485–486. IEEE, September 2008.

[MPP08b] M. Morandini, L. Penserini, and A. Perini. Modelling Self-Adaptivity: A Goal-Oriented Approach. In *SASO '08: Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 469–470. IEEE, October 2008.

[MPP09] M. Morandini, L. Penserini, and A. Perini. Operational Semantics of Goal Models in Adaptive Agents. In *AAMAS '09: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, pages 129–136. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[NE00] B. Nuseibeh and S. Easterbrook. Requirements Engineering: A Roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 35–46. ACM, 2000.

[ON98] P. D. O'Brien and R. C. Nicol. FIPA – Towards a Standard for Software Agents. *BT Technology Journal*, 16(3):51–59, July 1998.

[OPB00] J. Odell, H. V. D. Parunak, and B. Bauer. Extending UML for Agents. In *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, pages 3–17, 2000.

[PBL05] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI Reasoning Engine. In R. H. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 149–174. Springer US, 2005.

[PH07] M. P. Papazoglou and W.-J. v. d. Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, March 2007.

[PMC+12] G. Perrouin, B. Morin, F. Chauvel, F. Fleurey, J. Klein, Y. L. 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, 2012.

[PMR+07] C. Ponsard, P. Massonet, A. Rifaut, J. F. Molderez, A. Van Lamsweerde, and H. Tran Van. Early Verification and Validation of Mission Critical Systems. *Formal Methods in System Design*, 30(3):233–247, 2007.

[PMS08] A. Plsek, P. Merle, and L. Seinturier. A Real-Time Java Component Model. In *ISORC '08: Proceedings of the 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing*, pages 281–288. IEEE, May 2008.

[PPS+07] L. Penserini, A. Perini, A. Susi, M. Morandini, and J. Mylopoulos. A Design Framework for Generating BDI-agents from Goal Models. In *AAMAS '07: Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 610–612. ACM, 2007.

[PS11] L. Pasquale and P. Spoletini. Monitoring Fuzzy Temporal Requirements for Service Compositions: Motivations, Challenges and Experimental Results. In *RESS '11: Proceedings of the 2011 Workshop on Requirements Engineering for Systems, Services and Systems-of-Systems*, pages 63–69. IEEE, August 2011.

[PW03] L. Padgham and M. Winikoff. Prometheus: A Methodology for Developing Intelligent Agents. In F. Giunchiglia, J. Odell, and G. Weiß, editors, *Agent-Oriented Software Engineering III*, volume 2585 of *Lecture Notes in Computer Science*, pages 174–185. Springer Berlin Heidelberg, 2003.

[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.

[Rao96] A. S. Rao. AgentSpeak(L): BDI Agents Speak out in a Logical Computable Language. In *MAAMAW '96: Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-agent World: Agents Breaking Away*, pages 42–55. Springer Berlin Heidelberg, 1996.

[RG95] A. Rao and M. P. Georgeff. BDI Agents: From Theory to Practice. In *ICMAS '95: Proceedings of the First International Conference on Multi-Agent Systems*, pages 312–319. AAAI Press, Palo Alto, California, USA, 1995.

[RS77] D. T. Ross and K. E. Schoman, Jr. Structured Analysis for Requirements Definition. *IEEE Transactions of Software Engineering*, 3(1):6–15, January 1977.

[SBW+10] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein. Requirements-Aware Systems: A Research Agenda for RE for Self-adaptive Systems. In *RE' 10: Proceedings of the 18th IEEE International Requirements Engineering Conference*, pages 95–103. IEEE, September 2010.

[Sch06] D. Schmidt. Model-Driven Engineering. *Computer*, 39(2):25–31, February 2006.

[SLB08] Y. Shoham and K. Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008.

[SLRM13] V. E. S. Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos. Awareness Requirements. In *Software Engineering for Self-Adaptive Systems II*, pages 133–161. Springer Berlin Heidelberg, 2013.

[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.

[SS14a] A. Sturm and O. Shehory. Agent-Oriented Software Engineering: Revisiting the State of the Art. In O. Shehory and A. Sturm, editors, *Agent-Oriented Software Engineering*, pages 13–26. Springer Berlin Heidelberg, 2014.

[SS14b] A. Sturm and O. Shehory. The Landscape of Agent-Oriented Methodologies. In O. Shehory and A. Sturm, editors, *Agent-Oriented Software Engineering*, pages 137–154. Springer Berlin Heidelberg, 2014.

[ST09] M. Salehie and L. Tahvildari. Self-Adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2, May):1–40, 2009.

[SVB+08] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, and I. Crnkovic. A Component Model for Control-Intensive Distributed Embedded Systems. In *CBSE '08: Proceedings of the 11th International Symposium on Component-Based Software Engineering*, pages 310–317. Springer, October 2008.

[SW07] J. Shore and S. Warden. *The Art of Agile Development*. O'Reilly, 2007.

[Szy02] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[TMA03] T. Tung Do, Manuel Kolp, and Alain Pirotte. Social Patterns for Designing Multiagent Systems. In *SEKE '03: Proceedings of the 15th Int. Conf. on Software Engineering and Knowledge Engineering*, pages 103–110. Knowledge Systems Inst, Skokie, Illinois, USA, 2003.

[TMC+13] C. Trubiani, I. Meedeniya, V. Cortellessa, A. Aleti, and L. Grunske. Model-based Performance Analysis of Software Architectures under Uncertainty. In *QoSA'13: Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*, pages 69–78. ACM, 2013.

[C09] A. Caplinskas. Requirements Elicitation in the Context of Enterprise Engineering: A Vision Driven Approach. *Informatica*, 20(3):343–368, August 2009.

[VL00] A. Van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, pages 5–19. ACM, 2000.

[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 *Formal Methods for Software Architectures*, volume 2804 of *Lecture Notes in Computer Science*, pages 25–43. Springer Berlin Heidelberg, 2003.

[VL04] A. Van Lamsweerde. Elaborating Security Requirements by Construction of Intentional Anti-Models. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 148–157. IEEE, May 2004.

[VL07] A. Van Lamsweerde. Engineering Requirements for System Reliability and Security. In *Software System Reliability and Security*, pages 196 – 238. IOS press, 2007.

[VL09] A. Van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. John Wiley and Sons, 2009.

[VLL00] A. Van Lamsweerde and E. Letier. Handling Obstacles in Goal-Oriented Requirements Engineering. *IEEE Transactions on Software Engineering*, 26(10):978–1005, October 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.

[WC01] M. Wooldridge and P. Ciancarini. Agent-Oriented Software Engineering: The State of the Art. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *Lecture Notes in Computer Science*, pages 1–28. Springer Berlin Heidelberg, 2001.

[WH11] M. Wirsing and M. Hölzl, editors. *Rigorous Software Engineering for Service-Oriented Systems*, volume 6582 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011.

[WHS06] T. D. Wolf, T. Holvoet, and G. Samaey. Development of Self-organising Emergent Applications with Simulation-Based Numerical Analysis. In S. A. Brueckner, G. D. M. Serugendo, D. Hales, and F. Zambonelli, editors, *Engineering Self-Organising Systems*, volume 3910 of *Lecture Notes in Computer Science*, pages 138–152. Springer Berlin Heidelberg, 2006.

[Win05] M. Winikoff. Jack Intelligent Agents: An Industrial Strength Platform. In R. H. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 175–193. Springer US, 2005.

[WJK00] M. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, September 2000.

[Wol07] T. D. Wolf. *Analysing and Engineering Self-Organising Emergent Applications*. PhD thesis, Katholieke Universiteit Leuven, May 2007.

[Woo97] M. Wooldridge. Agent-based software engineering. *Software Engineering. IEE Proceedings*, 144(1):26–37, February 1997.

[WRH+12] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering*. Springer-Verlag Berlin Heidelberg, 2012.

[WSB+10] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel. RELAX: a language to address uncertainty in self-adaptive systems requirement. *Requirements Engineering*, 15(2):177–196, March 2010.

[WSB11] K. Welsh, P. Sawyer, and N. Bencomo. Towards Requirements Aware Systems: Run-time Resolution of Design-time Assumptions. In *ASE '11: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 560–563. IEEE, November 2011.

[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.

[YLL+08] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos, and J. C. S. P. Leite. From Goals to High-Variability Software Design. In *Foundations of Intelligent Systems*, volume 4994 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2008.

[Yu95] E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, Graduate Department of Computer Science, University of Toronto, 1995.

[Yu97] E. Yu. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In *RE'97: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, pages 226–235. IEEE, 1997.

[Zav97] P. Zave. Classification of Research Efforts in Requirements Engineering. *ACM Comput. Surv.*, 29(4):315–321, December 1997.

[ZSL14] P. Zoghi, M. Shtern, and M. Litoiu. Designing Search Based Adaptive Systems: A Quantitative Approach. In *SEAMS '14: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 7–16. ACM, 2014.

Web References

[1] ASCENS: Autonomic Service-Component Ensembles.
www.ascens-ist.eu

[2] CDEECo.
https://github.com/d3scomp/CDEECo

[3] Computing Research and Education Association of Australasia (CORE). The CORE Conference Ranking.
http://core.edu.au/index.php/categories/conference%20rankings/1

[4] DAUM project.
http://daum.gforge.inria.fr

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

[6] Eclipse. Eclipse Modeling Framework.
https://www.eclipse.org/modeling/emf/

[7] Eclipse. Epsilon language family and tool suite.
http://www.eclipse.org/epsilon/

[8] Eclipse. Graphical Modeling Framework.
https://wiki.eclipse.org/Graphical_Modeling_Framework

[9] ECSEL Joint Undertaking. ECSEL Multi-Annual Strategic Plan, ECSEL-GB-2014.22, 2015.
http://www.ecsel.eu/web/downloads/Documents_GB/ecsel-gb-2014-22_masp_2015.pdf

[10] European Union Horizon 2020, Smart Cyber-Physical Systems, ICT-01-2014.
http://ec.europa.eu/research/participants/portal/desktop/en/opportunities/h2020/topics/78-ict-01-2014.html

[11] Google. Google Guice.
http://code.google.com/p/google-guice/

[12] i* Quick Guide.
http://istar.rwth-aachen.de/tiki-index.php?page=iStarQuickGuide

[13] IRM-SA empirical study. Department of Distributed and Dependable Systems.
http://d3s.mff.cuni.cz/projects/components_and_services/irm/empiricalstudy/

[14] IRM-SA editor.
https://gitlab.d3s.mff.cuni.cz/iliasg/irm-sa-tool

[15] IRM-SA jDEECo plugin.
https://github.com/d3scomp/IRM-SA

[16] ISO/IEC/IEEE 29148:2011(E) – Systems and software engineering – Life cycle processes – Requirements engineering, December 2011.
http://dx.doi.org/10.1109/IEEESTD.2011.6146379

[17] JACK Intelligent Agents, Autonomous Decision-Making Software.
http://aosgrp.com/products/jack/

[18] Jadex.
http://www.activecomponents.org/

[19] jDEECo.
https://github.com/d3scomp/JDEECo

[20] jRESP: Java Runtime Environment for SCEL Programs.
http://jresp.sourceforge.net/

[21] Kevoree.
http://kevoree.org/

[22] MODELS 2014. Call for ACM Student Research Competition.
http://models2014.webs.upv.es/acmsrc.htm

[23] National Science Foundation, Cyber Physical Systems, nsf15541.
http://www.nsf.gov/pubs/2015/nsf15541/nsf15541.htm

[24] Object Management Group. CORBA Component Model Specification v4.0.
http://www.omg.org/spec/CCM/

[25] Oracle. Enterprise JavaBeans specification v3.2.
http://jcp.org/aboutJava/communityprocess/final/jsr345/index.html

[26] OSGi Alliance. OSGi service platform, core specification, release 5.
http://www.osgi.org/Specifications/HomePage

[27] OSGi Alliance. OSGi service platform, core specification, release 5.
http://www.osgi.org/Specifications/HomePage

[28] RELATE: Trans-European Research Training Network on Engineering and Provisioning of Service-Based Cloud Applications.
http://www.relate-itn.eu/

[29] Sat4j: The Boolean satisfaction and optimization library in Java.
http://www.sat4j.org/

[30] SpringSource. Spring Framework.
http://www.springsource.org/

[31] Tropos project – research areas.
http://www.troposproject.org/node/276