Charles University in Prague

Faculty of Mathematics and Physics

DOCTORAL THESIS

Michał Kit

Component-based engineering of Smart Cyber-Physical Systems

Department of Distributed and Dependable Systems

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

Prague 2016

Acknowledgement

I thank all who have supported me throughout my doctoral studies, and who have made this work possible. First and foremost, I would like to express my deepest gratitude to both of my advisors: Tomáš Bureš and Petr Hnětynka. I thank Tomáš for all his mentoring, guidance and expertise he shared with me for the last four years. The essence of those is reflected in this thesis and in all my publications. To Petr, I am grateful for his support (research and other –wise) from the very first day of my studies. His assistance in introducing me to the group, university and other aspects related to living in a new country has made out of this a great experience. I am deeply grateful also to František Plášil for his advice, collaboration, and dedication, especially during joint paper writing.

I thank my closest co-researchers Ilias Gerostathopoulos, Jaroslav Keznikl and Rima Al Ali with whom I shared most of the time during this endeavor and on whom I could always count.

My gratitude goes also to my colleagues at the Department of Distributed and Dependable Systems for their support and for creating such a positive working environment. In particular, I would like to thank Andranik Muradyan, Dominik Škoda, Filip Krijt, Jakub Daniel, Jan Kofroň, Jiří Vinárek, Martin Děcký, Paolo Arcaini, Pavel Jančík, Pavel Ježek, Pavel Parizek, Peter Libič, Petr Tůma, Viliam Šimko, Vladimír Matěna, Vojtěch Horký, and Zbyněk Jiráček. I also thank Petra Novotná for her unmatched positive attitude and support with all administrative matters.

All of this would not have happened without the financial support of the EU project RELATE 264840, being part of the Marie-Curie Initial Training Network of the 7th Research Framework Programme. To people behind it, I am also grateful for giving me an opportunity to exchange experiences with other researchers within the project and beyond.

Finally, I would like to thank everyone else outside research who has provided me with support and encouragement I needed during the last four years. Many thanks go to Olga and Viktor Fixl for sharing their home with me and creating a family-like environment.

Above all, though, I would like to thank my family who has supported me also before my studies. Especially, I thank my mother who has always been there for me with her unswerving encouragement.

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, February 1, 2016 ………………………

Michał Kit

Annotation

Title	*Component-based engineering of Smart Cyber-Physical Systems*
Author	Michał Kit kit@d3s.mff.cuni.cz (+420) 221 914 236
Department	Department of Distributed and Dependable Systems Faculty of Mathematics and Physics Charles University in Prague
Advisor	Doc. RNDr. Tomáš Bureš, Ph.D. bures@d3s.mff.cuni.cz (+420) 221 914 236
Mailing address	Department of Distributed and Dependable Systems Charles University in Prague Malostranské náměstí 25 118 00 Prague, Czech Republic
WWW	http://d3s.mff.cuni.cz/

Abstract

Smart Cyber-Physical Systems (SCPS) are distributed, open-ended and architecturally dynamic systems composed of autonomic components interacting with each other in cooperative actions and introducing system-level (emergent) behaviors that would not be possible otherwise. Very often, the components of SCPS vary with respect to purpose, behavior, and available resources.

Such characteristics of SCPS components (especially their heterogeneity combined with co-operativeness) allow for an overall resilience of the system as well as its continuous operation – the key properties the satisfaction of which is expected from any distributed system developed nowadays. Since SCPS is a relatively novel concept, there is no support in terms of design and development tools that would facilitate their engineering process.

This work aims to provide methods that address: development, verification and deployment stages of that process. In particular, the thesis focuses on delivering: (i) appropriate abstractions for SCPS modeling realization; (ii) a runtime environment for their deployment and execution; (iii) a simulation tool allowing for system-level verification. Altogether, they contribute to the DEECo framework, which is built around the DEECo component model.

Keywords

Software architecture, Component model, Ensemble-based systems, Gossip protocol, Cyber-Physical Systems, Simulation

Anotace

Název práce	*Komponentový vývoj Smart cyber-physical systémù*
Autor	Michał Kit kit@d3s.mff.cuni.cz (+420) 221 914 236
Katedra	Katedra distribuovaných a spolehlivých systémů Matematicko-fyzikální fakulta Univerzita Karlova v Praze
Školitel	Doc. RNDr. Tomáš Bureš, Ph.D. bures@d3s.mff.cuni.cz (+420) 221 914 236
Adresa	Katedra distribuovaných a spolehlivých systémů Univerzita Karlova v Praze Malostranské náměstí 25 118 00 Praha
WWW	http://d3s.mff.cuni.cz/

Abstrakt

Smart cyber-physical systémy (SCPS) jsou distribuované otevřené systémy s dynamickou architekturou. Tyto systémy jsou složené z autonomních komponent, jejichž interakce skrze kooperativní akce dává vzniknout komplexním chováním celého systému (emergent behaviors), jež by jinak nebyly možné. Komponenty, ze kterých se jednotlivý SCPS skládá, jsou velmi často rozdílné co do účelu, chování i dostupných zdrojů.

Tyto charakteristiky SCPS komponent (zejména kombinace jejich různorodosti a kooperace) přispívají k celkové odolnosti (resilience) takovýchto systémů, stejně jako k jejich souvislému fungování (availability) - což jsou klíčové vlastnosti, jejichž maximální splnění je očekáváno od moderních distribuovaných systémů. Jelikož jsou SCPS poměrně nový koncept, není pro ně podpora ve formě nástrojů pro návrh a vývoj, které by umožnily systematický inženýrský proces jejich tvorby.

Cílem této práce je poskytnout metody, které se soustředí na fáze vývoje, verifikace a nasazení v rámci takového procesu. Zejména si práce klade za cíl poskytnout následující: (i) vhodné abstrakce pro modelování SCPS; (ii) běhové prostředí pro jejich nasazení a operaci; (iii) simulační nástroj umožňující verifikaci na systémové úrovni. Tyto prvky dohromady jsou dodány coby součásti frameworku DEECo, který je postaven na bázi komponentového modelu DEECo.

Klíčová slova

Softwarové architektury, Komponentový model, Ensemble-based systémy, Gossip protokol, Cyber-physical systémy, Simulace

Contents

Chapter 1

Introduction

1.1 Smart Cyber-Physical Systems

Advancements in cloud computing together with an increasing number of interconnected devices alter the way we perceive and use things that surround us. More and more ideas are conceived every day, creating usage scenarios and business models, that few decades ago would not be possible. With the ongoing development in hardware electronics, which continuously get miniaturized, less energy-demanding, and most of all cheaper, successively we find new areas for their application. Moreover, a relatively low cost of electronic device interconnection allows for interfacing cars, washing machines, microwave ovens or even much simpler devices such as lighting bulbs and socket plugs. Ericsson predicts that the number of Internet-connected devices will grow 10 times from now on reaching the number of 50 billion in 2020 [Eri11]. In a similar vein, the SAP foretells the fourth industrial revolution, where technology merges physical and digital worlds connecting systems, networks and machines to enable a more autonomous and self-organizing approach to production [Ber13]. Therefore, more devices are

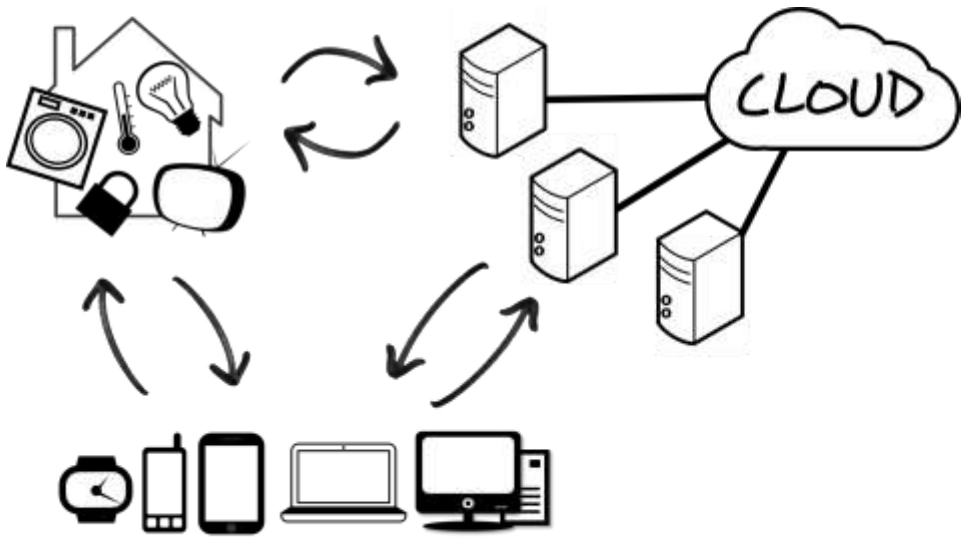


Figure 1. Connectivity in the Internet of Things.

getting manageable from any point in the world being agnostic to user's location. This emerging phenomenon of device interconnectivity is often termed as Internet of Things (IoT) [AIM10, McE14]. The WhatIs.com [1] defines IoT as "a scenario in which objects, animals or people are provided with unique identifiers and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction…". The "things" are no longer fully-fledged computers but devices equipped with fewer resources and providing much simpler functionality – e.g. sensors and actuators.

Being able to sense and actuate a physical property of a deployment environment is the domain of cyber-physical systems (CPS) [Bro13, LS10], which can be defined as "a system of collaborating computational elements controlling physical entities" [ML15].

While IoT focuses on devices interconnectivity and accessibility, CPS concentrate more on relations between cyber and physical worlds. Those two research domains, however, blend and complement each other providing for distributed systems that are highly interconnected, accessible and capable of altering the environment they *"live"* in. A home control system (also referred as "Smart Home") is one of the examples of this blend, where home appliances, sensors, and actuators (the CPS part) are steered remotely (the IoT part) to ensure safety and provide for user's comfort.

From the perspective of system architecture, each device can be seen as a component that interacts with other components in the system via connectors. As such, referring to the home control example, each home appliance is a separate component that interacts with a user or his personal device also represented as a component. There, the interactions and connectors are fairly simple and fixed. In nowadays emerging systems, however, we observe more sophisticated component interactions. Connectors are no longer static as in the aforementioned case, but they appear and disappear during the system lifetime. This stems from the deployment infrastructure, which is usually built over a wireless network, characterized by the unreliability of the communication medium (i.e. radio waves). In addition, to accommodate more devices and support new functionalities, such systems need to be open-ended allowing components to join and leave at any time, making the system (at least theoretically) unlimited with respect to the number of its components. The last cause for component connector transiency comes from system-level design, which assumes that components interact with each other only in some particular circumstances (i.e. in an ad-hoc manner). Since connectors embody those interactions, once the goal (specified at the design time of the system) of such interaction is achieved the connectors disappear.

An example illustrating this case comes from automotive industry and considers interconnected vehicles, cooperating with each other in order to optimize parking space selection process, which subsequently allows for an overall journey time reduction. Assuming distributed deployment of such system, it is not possible for a single component (in this case a vehicle) to have a global view on the system state. Thus, the components need to organize themselves into collaborative groups, delineated by some property of

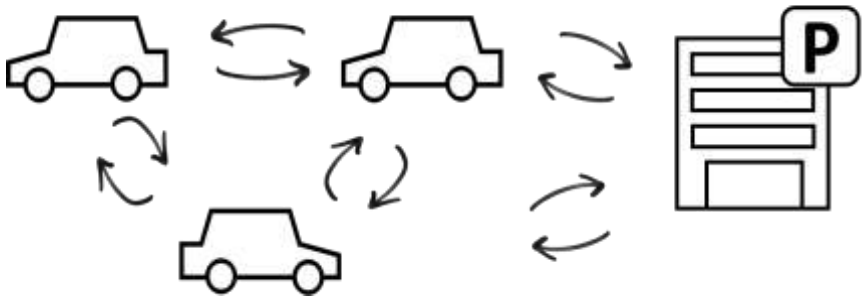


Figure 2. An example of vehicles coordinating with each other in order to optimize the parking place selection process.

the system (e.g. geographical proximity). Within such groups, they would then exchange necessary data (e.g. occupancy of already visited parking spaces) being used for selecting a parking space.

The cooperativeness becomes crucial not only when considering the semantic level of the system purpose but also when we take into account heterogeneity of devices, in terms of their hardware and available resources. Mobile phones, single-board, and personal computers, or large units used in data centers create a full spectrum of devices that vary not only in usage profiles but also in terms of available processing power and energy consumption. Thus, the need for computation off-loading from less capable hardware to one that is underutilized and equipped with more resources becomes pressing especially when considering resource sharing in order to secure, optimize and prolong system operability.

In addition, components themselves can contribute to the aforementioned system operability by making it more resilient to unexpected situations or situations regarded as abnormal. As such, components may implement different self-adaptation techniques [CCG+09, GSC09] that allow them to adjust autonomously their behavior (based on both self- and context- awareness [VH15]) to the current state of the system (or its observable part) and possibly undertake actions to restore its normalcy (i.e. its correct behavior with respect to system specification).

To sum up, the aforementioned aspects comprise the kind of cyber-physical systems that are highly interconnected, architecturally dynamic and composed of cooperative, heterogeneous components. Already mentioned, smart homes or smart vehicles are just a few examples of those. Because of their application in many smart-* cases, we will refer to this kind of systems as **smart cyber-physical systems** (**SCPS**).

1.2 Communication in SCPS (Challenges)

With the advent of SCPS, there is a demand for methods and tools that would address the SCPS entire lifecycle starting at the early stage of requirements gathering and ending at the product maintenance phase. "Old methods" (such as classical component-based software engineering) in software development that have been used to engineer distributed systems do not fit very well as they fail to cope with the new characteristics introduced by SCPS. As described previously, design-level architectural dynamicity, open-endedness, and unreliability of the communication infrastructure impose a new set of challenges unprecedented (at least in combination and in such a scale) before, that need to be addressed by proposing appropriate methods and tools facilitating the entire life cycle of an SCPS. In this work, we will focus on the category of challenges that relate mainly to the problems of inter-component communication and its realization in the context of SCPS.

The first challenge here is to come up with an adequate representation of an SCPS component and a formal description of its interactions with other components. Such abstractions need to correspond semantically to the aforementioned nature of the SCPS, meaning mainly autonomy of components and transiency of their interactions. Moreover, they need to account for open-endedness of SCPS reflected by having no assumptions with respect to system scale.

Furthermore, in order to be able to decide with whom a particular component should interact, we need to have at least a partial view on the system – i.e. information about other components. Then based on their state, we can decide whether two components should interact or not. In other words, a state of a component needs to drive the decision on creating or removing a connector between two components.

As an illustration, let us consider the example from the previous section and assume that vehicles there interact with each other based on the distance from their destination. Specifically, a vehicle should interact (i.e. exchange information on parking space occupancy) only with vehicles, which are in a proximity (given by some arbitrary value) to the vehicle's destination, as they are likely to possess most recent information about parking space occupancy in the area. However, to fulfill the "*proximity*" condition, we need to know about other vehicles in the system and their geographical location, which is expressed in the state of a vehicle component.

Ideally, each component has an access to the global, a relatively recent view of the system (i.e. state of all other components), which is possible if the deployment infrastructure provides some guarantees over the communication reliability. In such cases, the access to the global state can be realized either in a centralized or distributed manner. SCPS, however, are built over wireless infrastructures, which provide no guarantees over the communication, as the communication medium there (i.e. a radio channel) is limited in range and prone to electromagnetic interference. Moreover, the mobility of components reduces the duration of communication links, which adds to the overall instability of the inter-component connectivity.

In the end, we get a highly unreliable environment where any network-level protocols (e.g. Ad-hoc On-Demand Distance Vector Routing [PR99], Dynamic Source Routing [JM96]) are of no use as they fail to cope with infrastructural dynamism. For the same reason, in such settings, any centralized solution is out of consideration as well. Therefore, another challenge in the context of SCPS realization is to disseminate component state across the network in a distributed manner assuming nothing about network configuration as well as its participants and at the same time provide for system correctness in correspondence to its requirements.

The last challenge identified in this work, assuming the aforementioned setup, is to provide means for analyzing SCPS with respect to its system requirements. Very often, SCPS, apart from general system requirements, may require providing guarantees for some of its non-functional properties (e.g. timely reaction to an event in the system). Staying in the context of the vehicles example, we could imagine a simple scenario consisting of a leader and follower, where one vehicle automatically follows another vehicle ahead. In this case, components (i.e. vehicles) need to exchange information about their position, speed, acceleration, etc. in order to maintain a safe distance between each other and avoid collisions. To ensure that system is correct and its implementation corresponds to the requirements drawn during the design phase of the system, we need a method that would verify the system with respect to its specification and possibly suggest a configuration (if such exists) that would ensure satisfiability of its requirements. Again, taking into account the nature of the deployment infrastructure, this is not an easy task to achieve, as the method needs to account for peculiarities of the eventual communication model and the aforementioned characteristics of the underlying network.

To sum up, there are three main challenges identified in the context of inter-component communication that are the focus of this work and are defined assuming the distribution of components within an SCPS deployed over an unreliable network infrastructure.

- **C1 – Abstraction & Semantics.** In order to support design and development of SCPS, adequate abstractions for components and their interactions are required. They need to be tailored to support component autonomy and transiency of component interactions.

- **C2 – Realization**. SCPS components, in order to interact with each other, have to have an access to an up-to-date and ever changing system view (i.e. know the state of other components in the system) or, at least, its relevant parts. This would then drive the decision-making on component interactions creation and disposal.

- **C3 – Analysis.** To support validation of SCPS with respect to its both functional and non-functional requirements, methods that facilitate this process are needed.

1.3 Research Goals

Keeping in mind the assumption of component distributed deployment across unreliable infrastructure; the thesis takes up the following goals as means of contributing to a solution addressing the challenges **C1**, **C2**, and **C3** from the previous section.

- **G1 – Component Model for SCPS development.** Propose a set of modeling abstractions for an SCPS component representation and its interaction (with other components) specification. The abstractions need to account for and support the autonomy of SCPS components as well as architectural dynamicity stemming from component connectors temporality.

- **G2 – Execution and Deployment Platform**. Propose a technique that apart from providing a deployment and execution environment for SCPS components would automate the component state dissemination process across other components, allowing for distributed decision making about interactions between them (as described in **C2**). The dissemination process should align with the continuous evolution of a component state, and account for heterogeneous network support. As such the following two specific goals are further formulated:

    - **G2a – Heterogeneous network support.** The method should provide support not only for infrastructure-based networks, where network-level protocols (e.g. IP) exists and can underlay the proposed data dissemination process but also networks of an ad-hoc nature, where no communication protocol is available (i.e. Mobile Ad-hoc NETworks – MANETs).

    - **G2b – Optimize Data Dissemination.** The whole technique should optimize the data dissemination process to account for limitations of the deployment infrastructure. Ideally, the optimization method shall account for the specifics of the underlying network as described in G2a. Since, in such settings, it is not possible for a component to have access to the global view of the system, the parts of it that are available to a component should be relevant with respect to their usability from the component's perspective.

- **G3 – Simulation framework for SCPS.** Deliver a method for assessing the correctness of a developed SCPS – i.e. correspondence to its both functional and non-functional requirements.

1.4 Structure

This work is delivered as a collection of published papers, which altogether describe the DEECo component model and its realization. In particular, the collection consists of computation and communication semantics of the model, its implementation and simulation environment allowing for system validation.

To further extend the scope of the papers and provide a unified view, the thesis gives a comprehensive description of the state-of-the-art in Chapter 2. It presents related work in the area of distributed system development. In particular, the chapter focuses on revising the existing software architectures (i.e. component-based and agent-based) and identifying their advantages and disadvantages in relation to their application in the context of the SCPS development (Sections 2.1.1 and 2.1.2). In the last subsection (i.e. Section 2.1.3), the ensemble-based architecture is introduced and its current realizations are again assessed against the SCPS implementation. The second part of Chapter 2 surveys over the well-known communication paradigms used in the context of distributed systems. The selection of the methods was based on their correspondence to the implementation part of the contribution, which included (or took into account) those approaches at any stage of the research and realization process. In the last part of Chapter 2, simulation-based verification methods are described and their applicability in the context of SCPS is assessed. Generally, Chapter 2 is structured to be in line with the research goals from Section 1.3.

Further, Chapter 3 provides a brief overview on the author's contribution considering challenges and goals specified in Section 1.3. It describes parts of the DEECo framework that the author has contributed to. In addition, the chapter consists of a short iteration over the main publications of the author, and their relevance with respect to the research goals.

Chapter 4 lists co-authored publications that detail more on the author's contribution briefed in Chapter 3.

Finally, Chapter 5 provides conclusions on the research delivered and presented in the thesis. It also consists some possibilities for future directions, continuing the work initiated by the author in the area of SCPS development.

Chapter 2

State of the Art

2.1 Software Architectures for SCPS

The following section gives an overview on available software architectures that are tailored for building distributed systems in general. Based on this brief survey, pros and cons of each of the approach are then summarized and its applicability to SCPS design and development process is assessed. The main purpose of this survey is to set a context for the goal G1 and provide argumentation on decisions made while proposing the solution (see Chapter 3).

2.1.1 Component-based Architectures

Broadly recognized and adopted component-based software architectures divide complex systems into smaller reusable parts that can be used across different applications. By separation of concerns, they provide means for building and maintaining both large and small software products, which are much easier to comprehend through all the stages of their lifecycle. The basic element in this approach is a *component*, which has earned many different interpretations and as such does not have a single unequivocal definition. Below, there are just a few such interpretations of a component available in literature:

"A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces." – Philippe Krutchen [BW98]

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to the third-party composition." – Clemens Szyperski [BW98]

"A component is a unit of distributed program structure that encapsulates its implementation behind a strict interface comprised of services provided by the component to other components in

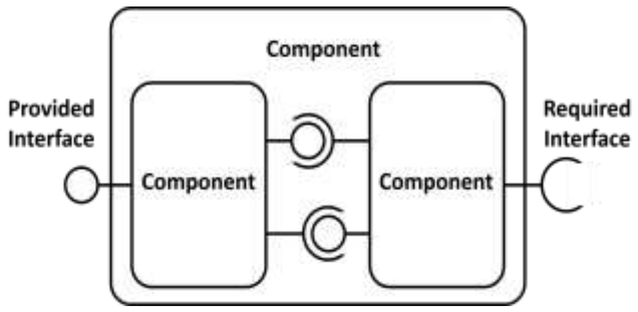


Figure 3. A composit component.

the system and services required by the component and implemented elsewhere. The explicit declaration of a component's requirements increases reuse by decoupling components from their operating environment." – Steve Crane and Nat Pryce [PC98]

From the architectural perspective, the component is described as a piece of software that encapsulates some functionality, hides its exact implementation from the rest of the world and provides its behavior through so-called *interfaces*. Each component may have two types of interfaces, which can be either provided or required. Provided interfaces are used to deliver functionalities to other parts of the system while required ones describe a functionality needed by the component implementation. From the outside world, a component is perceived as a black box, which communicates with other components through its interfaces, being the only interaction points with that component. This way, while designing an application we can focus only on functionalities disregarding all implementation related aspects and component internal architecture. Components can be composed hierarchically, defining a new component that aggregates functionalities delivered by each of its inner building blocks (see **Figure 3**). This kind of perspective or view on a component, where its internal structure is exposed, is called *gray box* approach and is handy when dealing with a single component development.

To sum up, in Component-Based Software Engineering (CBSE) different aspects can be separated from each other and contained in logical boundaries, constituted by a component. This allows for simplifications in the entire software development process, beginning with the design phase and ending at final product maintenance.

2.1.1.1 Selected representatives

There are many different solutions, available nowadays, designed for component-based software engineering. They offer various kinds of utilities for facilitating the entire application development process as well as provide means for their efficient execution. In their basics, they rely on the elementary assumptions of component-based software engineering, such as component interfaces, which by definition need to be well specified.

Typical examples of component models, that implement most of the concepts of CBSE, are SOFA 2 [BHP06], Palladio [RBB+11] and Fractal [BCS04]. Those mostly academic component models support both vertical (i.e. component nesting) and horizontal component composition and rely on strong dependencies (expressed by their interface specification) between components. Conceptually, they build over static architectures, defined usually in a dedicated ADL (Architecture Description Language).

OSGi

An example of a technology that adopts the ideas of CBSE and allows for relaxation in terms of component architecture is OSGi (Open Services Gateway Initiative) [2] [Boc11]. It uses the approach of service components, which in contrast to the standard CBSE paradigms, does not require system architecture to be defined at the design time and assumes a component instance appearance and disappearance during system runtime. OSGi was initially designed to introduce modularization for Java-based applications via so-called *bundles*. It loosens the dependency between components, by supporting reaction mechanisms on both communication link creation and its deletion during the application execution time.

As OSGi is just a specification, throughout the time, it has earned multiple implementations such as Equinox [3] or Felix [4]. While those are centralized distributions of OSGi, there is also a distributed version called Distributed OSGi (DOSGi) with the implementation provided by Apache CXF [5]. OSGi has also been used as a ground for component models realizations that encapsulate parts irrelevant (considering CBSE) from the perspective of actual usage and making it more suitable for a component driven design and development – e.g. iPojo [EHL07].

Progress

Another solution that leverages on the idea of components and concentrates on embedded systems is Progress [HPB+10]. Due to the support of real-time aspects, Progress comes with both time and reliability related analyzes that allow for verification and validation of a modeled system early at the design phase of the system lifecycle. As a framework, it delivers both a component model called ProCom [BCC+08, Led15] and a set of tools supporting the entire development process of embedded real-time systems. ProCom is based on two perspectives, differing in the level of granularity of the system being modeled. The fine-grained view is given by ProSave, which deals with low-level and passive (i.e. its activity is triggered externally) hierarchies of components. At this level, the functionality of each component is given as a set of independently running services that component supports. The inter-component communication is realized via data and triggering ports. A ProSave service is decorated with a single group of input and (possibly) several output group ports. A higher level of reasoning in ProCom is provided by the ProSys view. It describes the coarse-grained perspective of the system composed of concurrently running (potentially) distributed subsystems that communicate via channels supporting multiple senders and receivers. A ProSys subsystem (or component) can deliver its functionalities in a timely manner (i.e. periodically) or in a triggered fashion and can be modeled as a ProSave component.

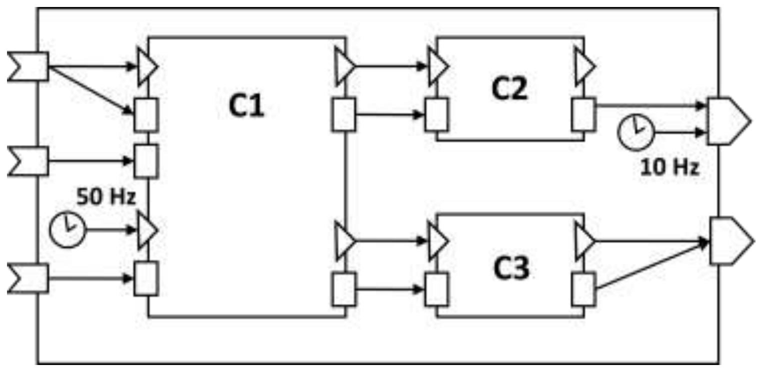


Figure 4. A ProSys subsystem internally modelled by ProSave.

SOFA HI

To support the development of component-based systems targeting embedded deployments, the SOFA HI (High Integrity) [PWT+08] has been proposed as an extension to the SOFA 2 [BHP06] component model. SOFA 2 itself builds on the hierarchical composition of components and is designed to support distributed system design, development, and deployment. SOFA 2 introduces the concept of the component repository, which manages SOFA 2 artifacts (i.e. components) and delivers them to the associated runtime environment. SOFA 2 components are described by the proprietary ADL that specifies both provided and required interfaces as well as the internal structure of the component (in the case of composites) or its realization given by an implementer (in the case of simple components). Moreover, each component in SOFA 2 is equipped with so-called micro-components, which encapsulate the control part of a component. Based on SOFA 2, SOFA HI is a constrained version of the original model, which is necessary in order to address the low-impact factor requirement in the embedded and real-time settings. At the same time, however, SOFA HI introduces extensions that are specific to this kind of deployment – i.e. specification of non-functional properties (e.g. timing aspects of component execution). Components in both SOFA 2 and SOFA HI communicate via connectors, which due to performance restrictions can by dynamically generated only in the original SOFA 2, supporting dynamic architectural reconfigurations. SOFA HI components are developed in C programming language aided by a set of development tools, including a dedicated IDE (Integrated Development Environment) and tools providing formal analysis and verification of the implemented system.

Kevoree

To mitigate the problem of architectural information centralization, the idea of models at runtime (models@run.time) [BFCA14] has emerged. The principle behind is that the data about the system architecture is encoded into the application model, which is then distributed across the system deployment units (i.e. physical nodes across which

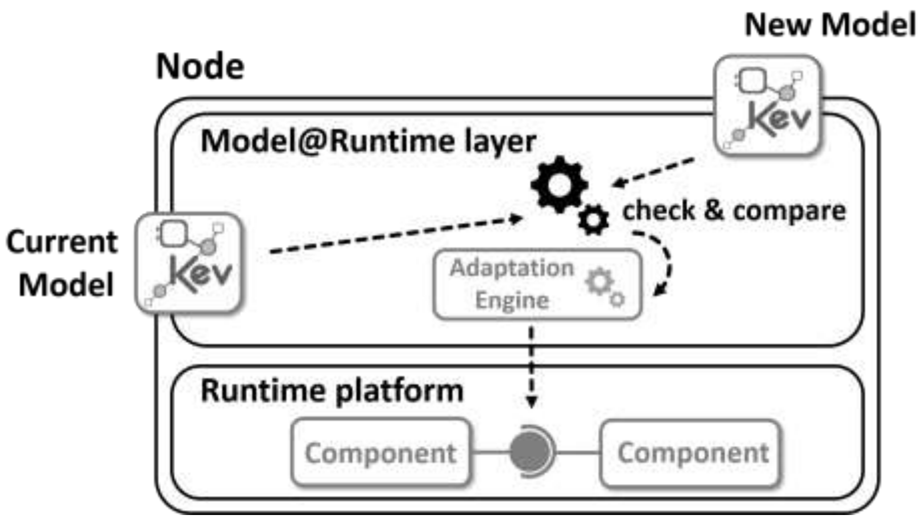


Figure 5. Kevoree models@run.time framework.

system is deployed). Whenever, a change to the application model occurs, the system triggers an adaptation mechanism that adapts its current state to the new one that reflects the most recent changes in the application architecture. The updated model is then synchronized across the system, which if required executes necessary actions that adjust its parts to stay in conformance with the current model. The key assumption here is that a change to the model may occur anywhere in the system, triggering a model update in the rest of it.

This approach has been implemented by the Kevoree framework [FMF+12] [6], which consists of the dedicated component model, deployment environment and a set of tools facilitating the whole development process and final product deployment. The elementary concepts in Kevoree are *component*, *channel*, *node* and *group*. While the component resembles components from other CBSE models, in Kevoree both the channel and node are seen as deployment abstractions. Channel is a component connector that allows for modeling different communication styles (i.e. synchronous or asynchronous) between components. Node, on the other hand, abstracts a component deployment unit. Nodes can be associated into groups, which delineate the application model consistency meaning that nodes of the same group have their application models synchronized.

While in Kavoree a single point of failure does no longer exists, as each node holds the information about the system architecture (i.e. the application model), the problem of strong inter-component dependency remains. Moreover, as every change into a system is followed by model adjustments and synchronization process, which are time-consuming (especially in the case of a large and complex system), there is a limitation with respect to the scale of system dynamism being supported. This prevents Kevoree from being used in scenarios, which include for instance node mobility.

2.1.1.2 Summary

CBSE is well-known and broadly used approach, which benefits, such as reusability or complexity reduction, have been recognized in the industrial software development, ranging from general-purpose applications to very specialized safety-critical embedded systems. This recognition comes from the advantages that CBSE brings to the software development, which are mainly separation of concerns and reusability.

There are few limitations, however, constraining CBSE methods application in the context of SCPS engineering process. The main issue is the assumption of centralized ownership and deployment of components. As mentioned in the above representatives, when building a component-based system, the usual scenario is to describe the system in terms of components, find appropriate component realizations (possibly in some repository) or implement them yourself. This holds in the case of systems that are distributed but the full control over the deployment environment is centralized and the system architecture is known beforehand. SCPS do not satisfy any of those conditions. By assumption, they are fully distributed, open-ended and their architecture changes continuously and unanticipatedly.

Another important drawback of CBSE is a strong inter-component dependency, which comes from the component interfaces that needs to be well specified. In order to support open-endedness of SCPS and future flexibility in terms of supported functionalities, the SCPS possible participants (i.e. components) stay unrestricted after the design time. As such, component dependencies cannot be fully specified at this point, which effectively makes CBSE techniques unsuitable.

Finally, CBSE requires some level of guarantees over the communication infrastructure that the system is deployed on. Network failures are allowed and tolerated but only to some respect. Again, this is violated in the context of SCPS. Most of the scenarios for SCPS assume mobility of components, which implies a high degree of unreliability considering communication link stability. In other words, SCPS adopts the idea of opportunistic (or best-effort) approach towards inter-component communication.

2.1.2 Agent-based Architectures

Another approach in building distributed systems is Agent-Oriented Software Development (AOSD). It relies on the concept of a *software agent*, which performs its actions on behalf of an external authority (i.e. host). Formally, "a software agent is a persistent, goal-oriented computer program that reacts to its environment and runs without continuous direct supervision to perform some function for an end user or another program" [1]. From outside, an agent can be perceived as an entity, which exhibits some degree of autonomic behavior and is capable of co-operating with other agents in order to achieve both individual and collective goals.

There exist different types of software agents, which may be classified according to their main traits like for example intelligent agents [Gil97], being able to learn and reason or distributed agents [Min98], which act on physically distinct machines. In terms of communication between agents, the most popular mechanism is messaging, which is utilized through programming abstractions provided for example by agent-oriented languages [ABH+06, BBD+06, Hin09].

Similar to component-based software engineering, the agent-based approach builds over the explicit communication, which means that the whole communication procedure needs to be handled by an agent, who explicitly decides when and whom to communicate with. In the case of agents, however, communication is more flexible (comparing to the standard CBSE) as a message recipient can be established during the application execution time, but still explicit addressing is required (see **Figure 6**). With that respect, they resemble service components (see the previous section).

Semantically, the difference between agent and component lays in the nature of interactions with their peers. Components provide a functionality to the system and are expected (usually) to be available through the entire lifecycle of an application. Their role is rather passive and constrained to serving requests from other components when needed. Agents, on the other hand, are active, and more autonomous in their actions.

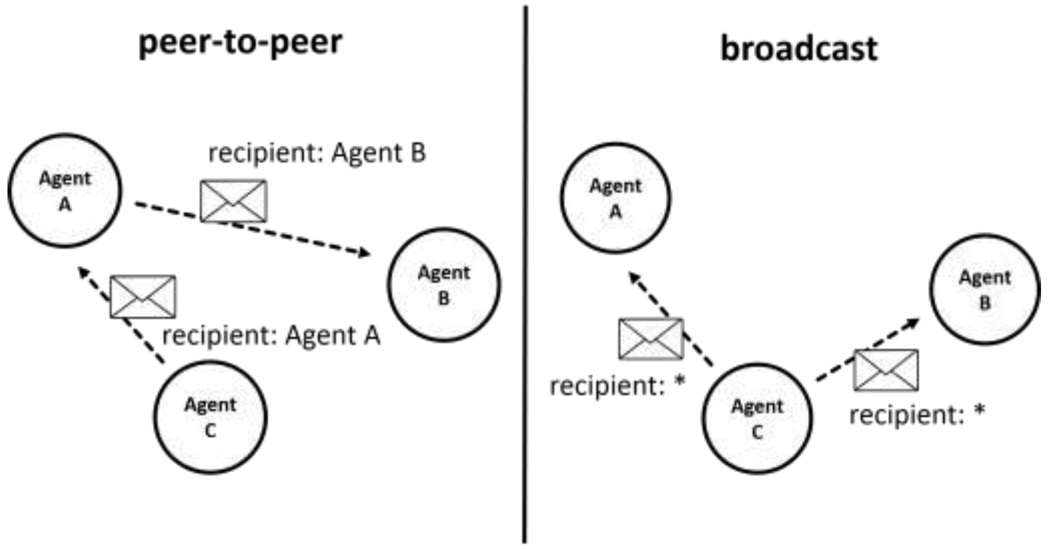


Figure 6. Messaging in agent-based systems.

Comparing to a component which is triggered by either event in the system or a request occurring on one of its provided interfaces, an agent is assumed to execute continuously and is capable of initiating interactions with other agents at any time of during its presence in the system. There is no strict dependency between agents that need to exist throughout the entire lifecycle of the system. As such, agents may appear and disappear from the system at any time, which results in more dynamic and resilient systems but this comes at the cost of complexity, which is the main issue when designing an agent-based system.

2.1.2.1 Selected representatives

BDI model

Being a focal point in the AOSD development process agents require a specialized approach that would support the autonomy and self-adaptation throughout the entire agent lifecycle. The well-known technique in this area is Belief-Desire-Intention (BDI) software model [PBL05, RG95]. Belief corresponds to agent's perception of the environment, which due to possible inaccuracies stemming from limited agent sensing capabilities is differentiated from the actual environment state. Agent's desires refer to its goals that the agent strives to achieve. Finally, the intentions are given in the form of plans containing a sequence of actions that would lead to achieving agent goals. The whole reasoning in the BDI model loops around sensing the environment (i.e. agent context) and forming a belief about it. Next, the analyzing step is triggered in which the agent's goals corresponding to the current situation are selected. Then the plans referring to those goals are followed by executing corresponding actions introducing changes to the agent's context. The whole reasoning cycle is illustrated in **Figure 7**.

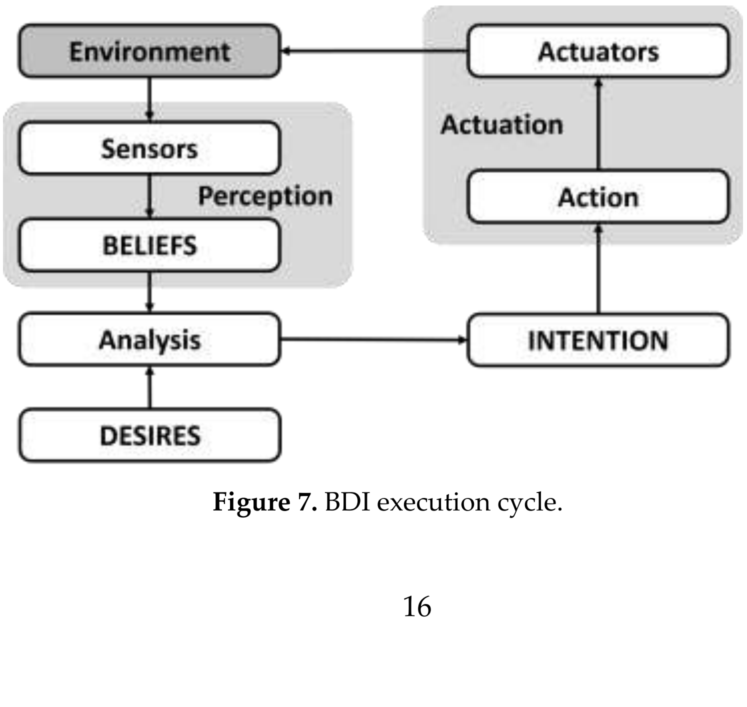


Figure 7. BDI execution cycle.

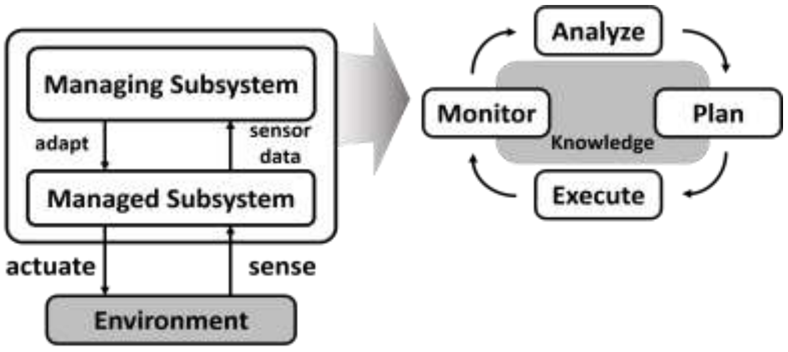


Figure 8. MAPE-K loop.

The BDI model conforms to the MAPE-K (Monitor, Analyze, Plan, Execute, and Knowledge) model [ARS15, IBM06], which serves as a general adaptation pattern used to engineer autonomic and self-adaptive systems [WSG+13]. As depicted in **Figure 8**, the MAPE-K model consists of a cycle, built out of four main steps. During the *monitoring* phase, information about the execution context is gathered, via sensors available in the system. Then, the data is *analyzed* and actions to be taken are *planned*. Finally, the actions are executed, potentially introducing changes to the context via a set of actuators.

While the MAPE-K model is a general prescription on performing the adaptation process, the BDI can be seen as its specialization, defining self-adaptation realized in the context of multi-agent systems.

Holons

Due to the high degree of autonomy, agents exhibit an interesting property of self-organization and collective behaviors [KGJ09]. Those two characteristics allow developers to build complex systems out of very simple agents, which, on their own, have no capabilities (or they are highly limited) to perform tasks given to the system. As such they need to organize themselves and cooperate by splitting bigger tasks into smaller

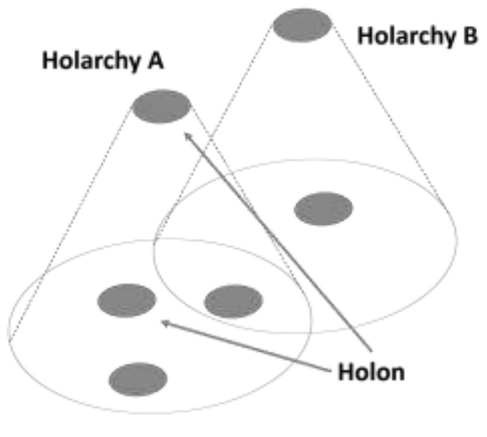


Figure 9. Holons and their holarchies.

ones (a.k.a. Divide and Conquer technique [ER94]) that can be executed by an individual agent.

The Holonic Multiagent Systems (HMS) [Fis99, FSS03] provide for whole theory and terminology that allows for reasoning about dynamic organizations of agents and their collective behaviors. The term *holon* stands for Greek "holos" translating to "whole" and suffix "-on" that means "part". The HMS changes the AOSD paradigm of an agent to a hierarchical structure called *holon*. The holon can be either a simple agent or a composition of agents. Sustaining the agent autonomy agents decide themselves, whether it is beneficial to become part of a holon or not. Moreover, agents can belong to multiple holons at the same time. The hierarchies of holons termed as *holarchies* introduce the concept of vertical composition, where each holarchy level (i.e. holon) is a composition of other holons and composes the upper-level of the holarchy (see **Figure 9**).

As such, in terms of the structural hierarchy, the HMS extends the approach of CBSE and composite components, with a possibility of an agent belonging to multiple super-agents (i.e. holons), which is not the case in CBSE where a component instance being a part of one super-component cannot compose another one.

Due to high complexity and a lack of adequate modeling abstractions, to this end, there exist only a few implementations [Fis98, GGHK09] of the HMS approach in the context of multi-agent systems and they are mainly used for simulation purposes.

2.1.2.2 Summary

The advantage of AOSD over CBSE lays in the autonomy of agents, easing the development of adaptive and more resilient systems. By design, agents are meant to execute in a collective, providing the system with emergent behaviors that would not be possible otherwise. Those characteristics are also desired in case of SCPS.

The main issue with AOSD is the aforementioned lack of adequate programming constructs that would map the concepts of autonomy, adaptation, and collective cooperation, into modeling constructs that would ease the whole design process. Because of this missing expressiveness, it is a complicated task to build a system composed of heterogeneous agents, which would be equipped with more than trivial logic.

Moreover, the complexity of the system is affected, due to explicit communication imposed by message-based approaches used in AOSD. Agents, in order to exchange messages, need to know the recipient of a message beforehand. This implies the presence of so-called the "*yellow pages*" service that would provide an agent with its peers in the system. Consequently, the whole approach is centralized and together with the overgrown complexity stemming from the aforementioned factors, usage of AOSD in the context of SCPS development is effectively constrained.

2.1.3 Ensemble-based Architectures

The Ensemble-Based System Engineering (EBSE) is a new method in design and development of dynamic distributed systems. It adopts some elements from the aforementioned component- and agent-based techniques. Researched under the EU FP7 project focusing on Autonomic Service Component Ensembles (ASCENS) [7], it introduces a set of tools dedicated to supporting the entire lifecycle of systems built of cooperative and autonomic components.

The core concepts behind the ensemble-based systems are *component* and *ensemble* of components. A component in the view of EBSE is an active and autonomous entity just like agents in AOSD. Nonetheless, it is still a unit of encapsulation that is reusable and replaceable. In EBSE approach, a component is composed of both knowledge, which is a set of attributes reflecting the component state and processes comprising the logic of the component. Processes execute continuously upon the component knowledge, reading and modifying its attributes and effectively altering the component state.

EBSE follows some of the CBSE development process rules – e.g. separation of concerns. Moreover, the concept of a component interface exists also in EBSE, and similarly to CBSE, defines a set of attributes provided externally by a component reifying the interface. Furthermore, interfaces are part of ensemble specification, which is a prescription on inter-component data exchange. An ensemble is formed dynamically depending on the state of components. Components are continuously monitored (i.e. their attributes exposed by interfaces) against ensembles membership. Those, belonging to an ensemble, exchange data between each other and based on that, undertake cooperative actions that lead to emergent behaviors in the system. The basic concepts of EBSE are depicted in **Figure 10**.

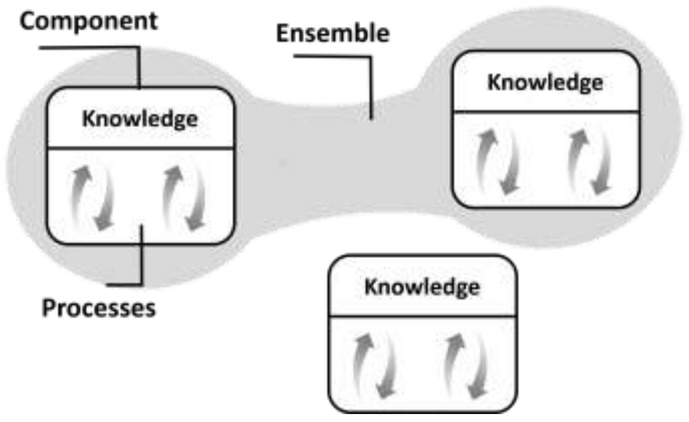


Figure 10. Components and ensembles in EBSE.

2.1.3.1 Selected representatives

SCEL

The cornerstone for EBSE has been laid by formulating Service Component Ensemble Language (SCEL) [DNFLP13] where the concept of the component ensemble was first introduced. SCEL is a formalism providing constructs for ensemble-based system specification and its main goal is to allow for system analyzes and verification at the early stage of the system lifecycle – i.e. system design and modeling. As a language, SCEL is built over few basic constructs, which mainly are component, knowledge, processes, interfaces, ensembles and policies. Since SCEL introduced most of those concepts to EBSE, their descriptions provided in the previous section remain valid also in the context of SCEL. The only extra item given in that list are policies [LMPT14], which are considered as an additional feature enriching the whole EBSE idea. In SCEL, communication between components is taken down to the level of component knowledge access (i.e. reads and writes of component attributes). Policies allow for restrictions in component knowledge access, which extends system analyzes by security related aspects.

SCEL has its realization in the Java framework called jRESP [8]. jRESP is an agent-based solution allowing to develop and execute SCEL components. Unlike another representative, it builds over an explicit form of communication, where agents (reifying the idea of SCEL components) initiate message-based data exchange process. The most flagrant case study developed in jRESP is the ASCENS Self-Aware Robots scenario simulating collaboration within a robot swarm executing in the context of a search and rescue mission [9].

HELENA

Driven by the same core concepts (i.e. component and ensemble) of the ASCENS project, the HELENA approach [HK14, KMH14] proposes an alternative view on the way we develop systems composed of autonomous components. While in SCEL (and

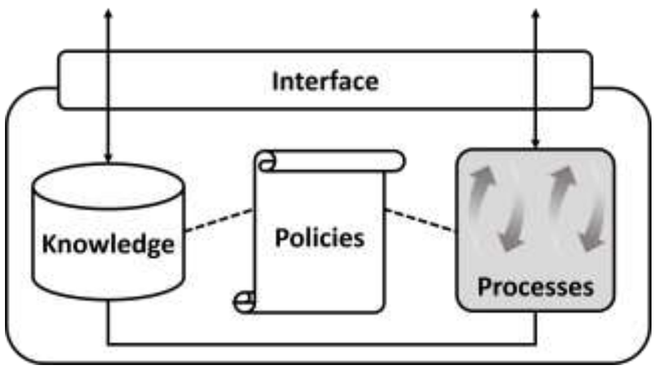


Figure 11. Component in SCEL.

jRESP) ensemble specification was blended into the components themselves, in HELENA ensembles are first class constructs that describe components involved as well as the whole communication flow between them. An HELENA component (or component type as types and instances are distinguished in HELENA) is given as a triple composed of a component name, state (being a set of attributes) and operations that the component support. Operations can be of the following types: internal (implementing component's autonomous logic) and two external (i.e. incoming and outgoing) encompassing bi-directional communication with other components. Each component may take several roles, which is a new construct (in EBSE) proposed by HELENA. A role is dynamically adopted by components and is used to decide on component's ensemble membership. It consists of a set of attributes required from the component, operations supported by the component and component types that are allowed to adopt this particular role. As a component can reify multiple roles, it can participate in multiple ensembles at the same time, which is also the case in SCEL.

An HELENA ensemble, on the other hand, is defined by roles (including their multiplicities), component types and connectors. The latter details about the communication flow, specifying the source, target, and format of that communication. On top of that, the ensemble definition is extended by its behavioral description given as a labeled transition system consisting of states defined by role attributes and transitions annotated by specific role operations. By externalizing communication from the logic of a component,

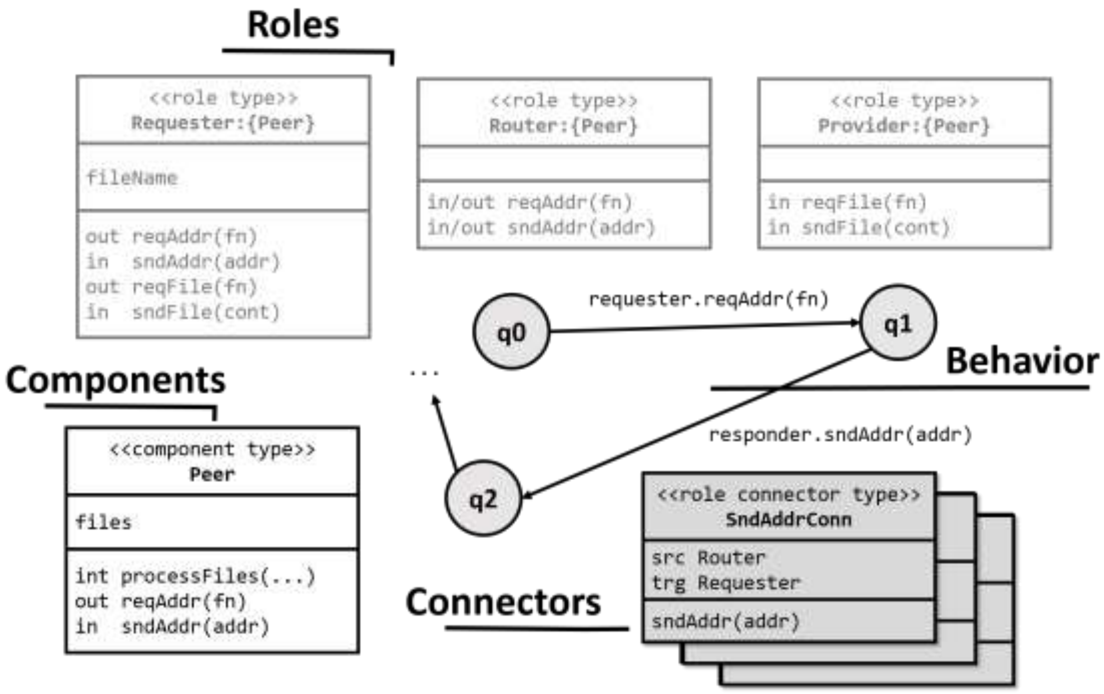


Figure 12. An overview of HELENA ensemble.

the whole process of data exchange between components becomes implicit, which simplifies considerably the design process of a system.

The HELENA approach has been validated in jHELENA [KCH15], which is an execution platform developed in Java. It supports system specification given in a DSL (Domain Specific Language) defined over HELENA abstractions, which are then translated into Java constructs that can be deployed and executed.

2.1.3.2 Summary

The EBSE delivers excellent means for modeling highly dynamic systems. By introduction of the idea of the ensemble as a description of temporal component grouping, it allows to express system dynamism in terms of logical predicates defined over component attributes. Moreover, defining components as active entities, EBSE assumes component autonomy, which resembles the AOSD approach (see Section 2.1.2).

Nevertheless, being a novel idea, EBSE lacks its proof of concept that would validate its general ideas and propose different realizations targeting especially real-life deployments. Existing solutions (i.e. SCEL and HELENA) omit technicalities such as those related to communication unreliability, which in the end affect the entire lifecycle of an SCPS.

2.2 Communication in SCPS

The following section gives an overview on different techniques that are currently adopted in the domain of distributed system communication. The examples presented here were selected as they served as an inspiration for building a solution that addresses goal G2 (see Chapter 3).

2.2.1 Message Passing

One of the basic paradigms in distributed system communication is Message Passing [10]. It relies on a simple idea of message exchange between two communicating peers. The sender of a message needs to be aware of its recipient unless the message is addressed to everyone who is reachable from the sender. In that case, we say that the message is *broadcasted*. Message broadcast, even though being the simplest way of data dissemination has been proved inefficient in case of networks with unreliable communication medium or with limited data bandwidth [TW11]. A better solution in that respect, that is able to cope with communication failures, is the Gossip protocol.

Gossip Protocol

The Gossip protocol [DGH+87, LPR10] is an algorithm that combines interval-based message sending together with message forwarding. Relying on periodic messaging and selective message propagation (i.e. the message is sent only to a subset of possible recipients, selected for instance probabilistically [GKM03]), the main issues of the broadcast (i.e. network flooding and proneness to network failures) are mitigated in Gossip. Adding message forwarding, applied by each recipient, data is effectively disseminated among all the recipients. There exist different variations of the Gossip protocol that are designed for infrastructure-based networks (e.g. LAN) [GKG02] but also for infrastructure-less deployments such as wireless environments [LM11]. The most distinguishing factor in Gossip implementations is the message sending interval calculation. Some use probabilistic-based period selection, other (for instance those designed for wireless net-

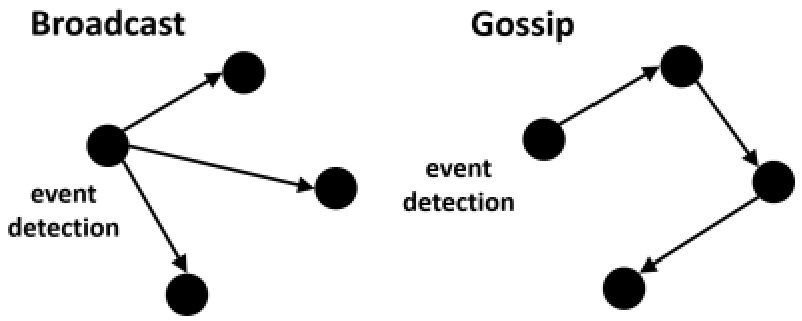


Figure 13. Broadcast vs Gossip protocol.

works) rely on signal strength when deciding how long to wait message (re) transmission. Furthermore, in wireless settings, there is a limitation on recipient selection only to those being currently in the radio communication range.

Wireless Sensor Network Protocols

Considering existing approaches that build over the idea of message passing, it is necessary to mention the Wireless Sensor Networks (WSN) [RSCB14]. This kind of networks is built over simple devices that are able to sense and actuate some physical properties of the environment that they are deployed in.

One of the prominent technologies used in this area is ZigBee [BPC+07], which is a communication protocol tailored specifically for WSN. It is built over the LowPAN (Low power Personal Area Network) [HC08] both physical and media access layer (MAC) protocol – IEEE 802.15.4 [SV08]. Features of the ZigBee protocol (considering its possible applicability in SCPS) fall into low energy consumption (comparing for instance to Wi-Fi), long communication range (up to 100m [BPC+07]) and short network association time (i.e. around 50ms), which becomes important when dealing with the mobility of network nodes. Those features, however, stem mainly from the lower stacks of the ZigBee protocol - essentially physical and MAC layers. The ZigBee network layer comes with a proprietary solution for maintaining logical networks of devices. It allows for different network topology organizations (i.e. start, mesh, cluster tree – see **Figure 14**), which are centered around a dedicated management node called coordinator. Therefore, as a coordinator is necessary for network creation, the ZigBee protocol turns out to be

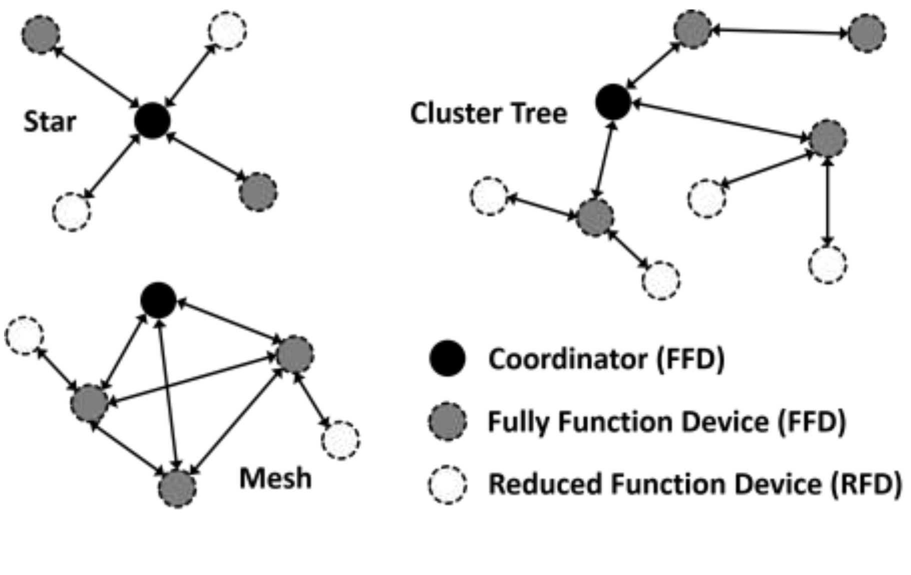


Figure 14. ZigBee network topologies.

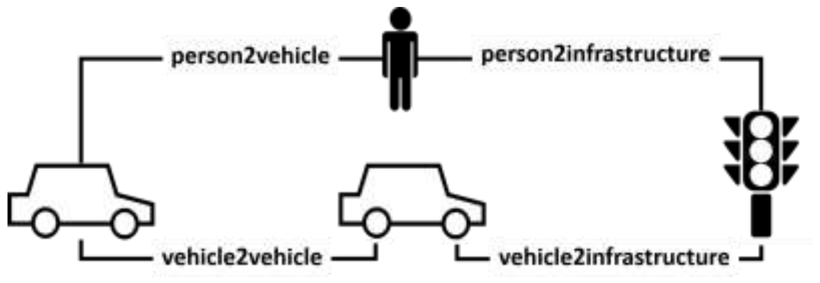


Figure 15. VANET communication.

limited with respect to ad-hoc scenarios, where devices establish a network without any prior infrastructure.

Vehicular Area Networks

From the perspective of SCPS, the Vehicular Area Network (VANET) [GSB12, KAE+06] is another interesting approach leveraging on the message-passing communication paradigm. Its differentiating factor is the assumption of node mobility and, more importantly, mobility with the assumption of a relatively high velocity. This, subsequently, translates to short network association times, which require dedicated techniques, being able to cope with such a requirement. Communication in VANET is of two sorts: infrastructure-based and infrastructure-less. The former allows vehicles to communicate with the roadside infrastructure (V2I), which is interconnected and serves as a network backbone for VANET. The latter one is formed between vehicles themselves (V2V) and relies on short-range radio transceivers mounted in the vehicles. It is characterized by low-latency, which makes it dedicated mainly for traffic carrying some critical information (e.g. emergency event detected by a vehicle ahead). The infrastructure-based network can also be used to provide an update on current road situation but additionally is meant to provision additional non-critical service data (e.g. weather or traffic updates).

VANET is an interesting example of combining the two kinds of communication realizations based on the aforementioned infrastructure-based and infrastructure-less networks. It seems to be an appealing approach also from the perspective of SCPS as the assumptions about the deployment environment are often similar – especially with respect to nodes mobility. Therefore, splitting the data dissemination into two types supported by different networking techniques, looks like a beneficial implementation strategy for SCPS communication. It gives more flexibility in data distribution comparing to only shared and limited short-range wireless communication usage.

2.2.1.1 Summary

The message-passing paradigm is close (in terms of the abstraction level) to the network-layer protocols of the OSI model [11]. By that, it provides a flexibility with respect to the communication protocols implementation. An example of such a protocol is the aforementioned Gossip protocol. However, in terms of its usability in the context of the SCPS development, the message-passing is too complex to be considered as a sole solution for component communication implementation. Since message delivery requires some prior knowledge about recipients (i.e. their addresses), it increases dramatically the complexity of the entire engineering process. As discussed in Section 2.1.2, one of the main drawbacks identified in AOSD preventing this approach to be implemented in the context of SCPS is the complexity that stems from the message-passing paradigm used in its pure form. Nevertheless, the message-passing paradigm remains usable as an underlying solution that could be hidden behind higher-level abstractions waiving away the complexity related to the message handling.

2.2.2 Publish-Subscribe

One of the most prominent technique in centralized message-passing is the Publish-Subscribe model [CSS11, She10]. The principle behind assumes a set of *producers* publishing messages to some centralized entity (usually called *broker*), and a set of *consumers* subscribing to that entity for messages of their interest. Usually, however, those two roles are combined into *prosumer* who can both produce and consume messages. The main advantages of the method are scalability allowing (theoretically) for an unlimited number of protocol participants and loose coupling which comes from the existence of the broker, which acts as a demarcation line between participants, often unaware of each other explicitly. The same broker, however, is the source of issues for the Publish-Subscribe. As participants of the communication protocol register with the broker their interest in some particular type of messages (usually expressed by message attributes), the whole mechanism imposes a strict message structuring, which in the end limits the flexibility of the method. Also, other limitations apply to broker configuration, which is to be chosen before system deployment, and it is meant to remain unchanged throughout the system lifetime.

There are multiple implementations of the Publish-Subscribe model, where most popular one is Java Messaging Service specification [12], being further embodied by ActiveMQ [13] or more recently developed for the purpose of the LinkedIn social service [14] – Kafka [15].

2.2.2.1 Summary

In terms of SCPS development, the Publish-Subscribe model becomes beneficial when considering deployment model that assumes some fixed, reliable and pre-existing infrastructure. An example could be a home LAN (Local Area Network) allowing for dynamic device association. In such a case, thanks to broker's loose coupling feature, SCPS components could join and disjoin the messaging system with no limitations. Nevertheless, being considered as a full-fledged and sole solution for SCPS is impossible due to broker's centralization.

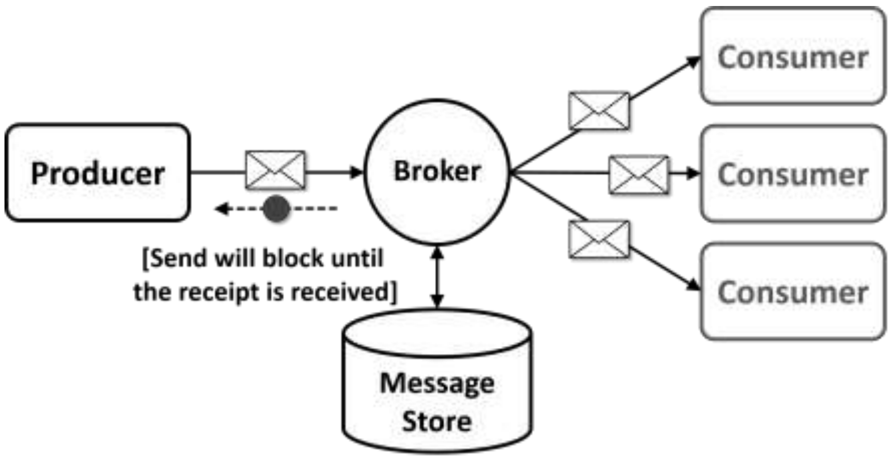


Figure 16. Publish-Subscribe.

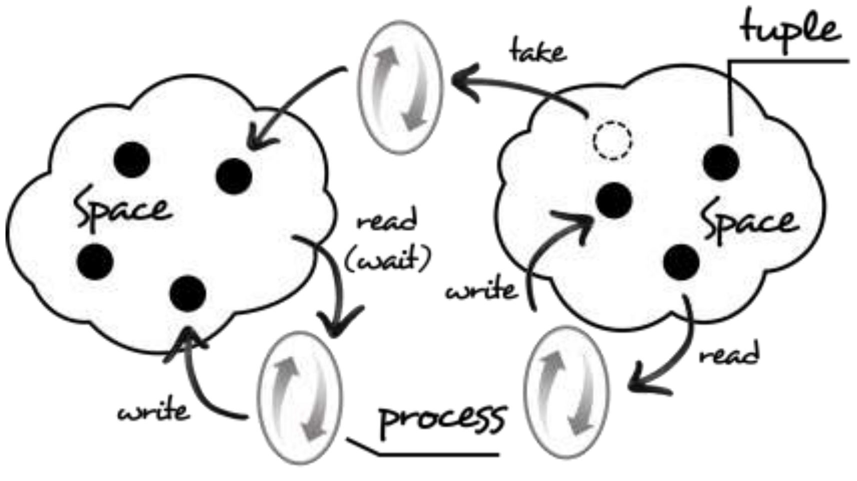


Figure 17. Tuple spaces.

2.2.3 Tuple spaces

The main idea behind tuple spaces [16] is to bring a unified and as simplest as possible view over memory, which is shared among different processes of some bigger system. It is a core concept of Linda languages [ACG86] that deal with the coordination problems between processes communicating over a shared memory. Processes are equipped with a set of operations that they can execute on the memory, which from their perspective is perceived as a black box. The operations include reading a tuple (i.e. a key-value pair) from a space – *read*, inserting a new tuple into a space – *write* and removing a tuple from a space – *take* (see **Figure 17**). This approach allows for separation of the internal realization of the shared memory (possibly distributed) from the application logic. Different implementations bring extensions (such as blocking or non-blocking operations) to the main principle of tuple spaces; however, the core idea stays the same.

Centralized Tuple Spaces

Considering centralized approaches towards Tuple Space, one of the well-known solutions is JavaSpaces [17] specification. JavaSpaces has set the ground for future realizations of the Tuple Space technique in the context of Java language. Its implementation is part of the Apache River project [18].

Other, centralized realization of the Tuple Space concept include GigaSpaces [19], SQLSpaces [20], SemiSpace [21]... Their common denominator is that the clients of a

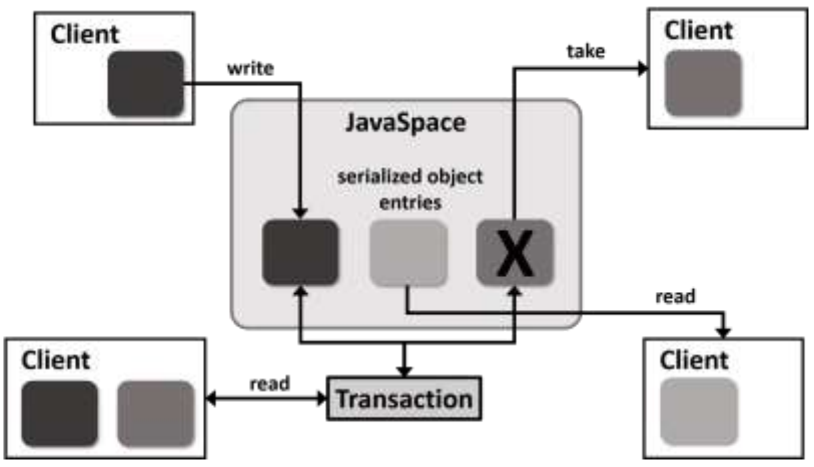


Figure 18. Centralized Tuple Space architecture.

space connect to some central point in the network and perform their actions remotely (e.g. via RPC - Remote Procedure Call). The general idea is illustrated in **Figure 18**.

Distributed Tuple Spaces

This is different comparing to what is understood under the distributed realizations of Tuple Spaces. In these cases, peers hold their own replicas of a space (or its part) and perform synchronizations based on periodic and reactive triggers (e.g. whenever clients are able to communicate). The principle applied here, is to disseminate data to everyone (see **Figure 19**), so there is a risk for an overhead imposed over the network infrastructure as (due to the lack of a central communication orchestrator) each network node needs to rebroadcast every data coming from any other node. To tackle this problem,

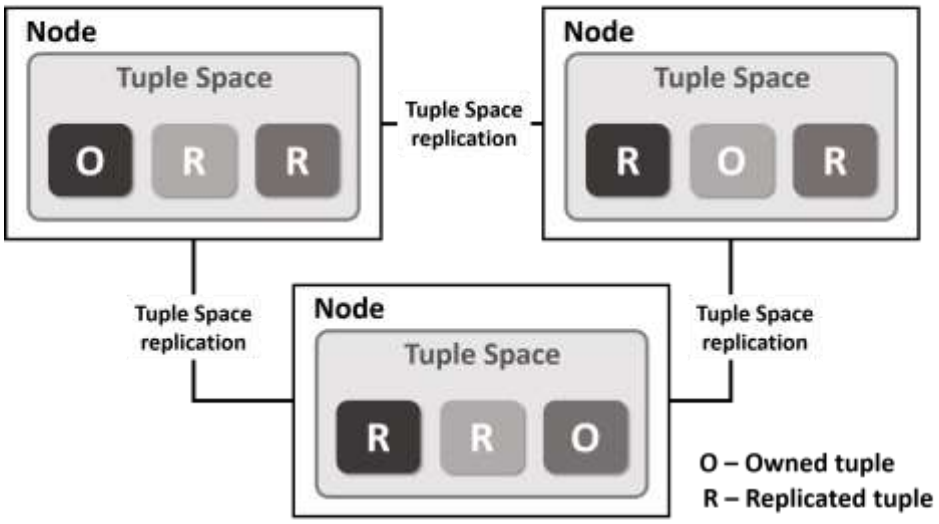


Figure 19. Distributed Tuple Space architecture.

few techniques reducing the amount of data propagated over the network has been proposed – e.g. through context awareness [CMMP09]. Examples of such approach implementers are EgoSpaces [JR06] or LIME (Linda In Mobile Environments) [MPR06].

2.2.3.1 Summary

Both approaches have their pros and cons. In the case of centralized solutions, the advantage is a lack of data replication, while their downside gets to the common issue of a single point of failure. In the case of distributed techniques, on the other hand, the main considerations are the aforementioned network overhead caused by data rebroadcasts and possible data desynchronization between its owner and replicas.

In the frame of SCPS development, both approaches become applicable depending on the deployment. While centralized implementations support performance and lesser resource utilization (e.g. network bandwidth, device local storage), the distributed ones go in hand with SCPS component autonomy and mobility.

2.3 Simulation frameworks for SCPS

This section covers the state of the art with respect to the goal G3. It gives a short discussion on currently available tools that facilitate verification process of a developed component-based system. In particular, it focuses on, assessing their applicability with respect to the SCPS validation.

As mentioned in the introductory section SCPS, unlike other component-based distributed systems, are meant to be open-ended and run in unpredictable settings. At the same time it is expected that the system remains resilient and its functionality available regardless to the aforementioned unpredictability. Thus, it becomes crucial to validate the system and assess its behavior with respect to various properties (e.g. network quality, component mobility etc.) of the deployment environment. To address this challenge, an approach based either on formal or experimental methods needs to be devised.

Formal methods are used to verify system model (most often at the early stage of the software engineering process – i.e. design phase) against some particular properties usually drawn from the system requirements specification. An outcome of such a verification delivers a mathematical proof on system correctness and can be used to certify correspondence of system implementation to its requirements. Despite being an undisputable (proof based on mathematical theorems) and inexpensive (i.e. any flaws in system design can be detected at the beginning of the development process) method, the formal verification suffers from the inability of system validation against multiple properties and simplified system models used during the process. This stems of course from exploded complexity, which grows together with the number of variables taken into account. Moreover, discrepancies in models descriptions defining system behavior (e.g. physical domains are usually expressed using continuous mathematics while computational or cyber elements are expressed using discreet one) add to the overall difficulty of the method [Lam05].

To overcome this issue, experimental methods provide a reasonable compromise between assurance level concerning the method outcome and the scope of properties that the system is validated against. The most common technique used in the experimental methods is a simulation. It allows for testing a developed system in simulated settings, which ought to resemble target (i.e. real-life) deployment. Unlike formal methods, simulations require a complete realization of a system (not only its model). They are usually executed multiple times under different scenarios that vary in deployment conditions reflected in the simulation configuration.

The focus of this thesis concentrates around final stages of the SCPS development process – i.e. realization and deployment. Since SCPS are relatively a novel concept it still lacks a tool support that would allow to position this work to. In terms of IoT simulators, there are first encounters (e.g. SimpleIoTSimulator [22]) trying to solve the problem of pre-deployment system validation using simulation-based techniques, however, they neglect some of the important aspects (e.g. detailed network models) throughout the simulation process. As such, in the remainder of this section, few examples of the simulation tools used in the (related) area of cyber-physical systems is presented.

CPS (and as such SCPS) combines two different domains: cyber and physical. Therefore, it is crucial that during the simulation process both of those domains are equally addressed. Until this date, there are not many simulation frameworks that would solve this issue holistically (i.e. provide comprehensive representations of the two domains). Those few that are there (e.g. TrueTime [HCeA03]), are limited in the scope and the level of details of the simulated domains.

There exist, however, so-called *co-simulation* frameworks that combine two (or more) simulation tools each addressing different aspects of the simulated environment.

PiccSim

PiccSim (Platform for integrated communications and control design, simulation, implementation, and modeling) [23] is an excellent example of a co-simulation platform that combines two well-known simulation tools: Simulink [24] that takes care of simulating control systems and ns-2 [IH08] for network simulations. Simulink on its own is a block diagram environment for multi-domain simulation and design of real-time embedded systems. Among its main features, the most notable are support for modeling, model-based simulation, automatic code generation, and continuous testing combined with verification of developed systems. Simulink facilitates the entire development process by providing a graphical editor, customizable block libraries, and a variety of solvers for dynamic systems modeling and simulation. Being integrated with MATLAB [25],

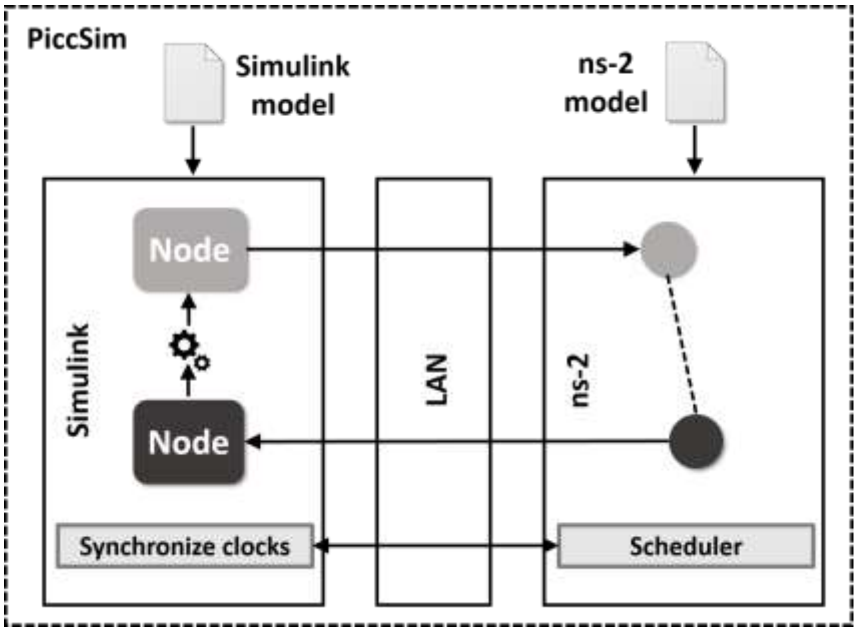


Figure 20. PiccSim architecture.

it enables incorporation of MATLAB algorithms into its own models and export simulation results to MATLAB for further analyzes.

ns-2, on the other hand, is a discrete-event network simulator that supports simulations of TCP, routing, and multicast protocols over wired and wireless networks. It provides detailed models for different communication protocols available nowadays in the networking domain.

Due to this synergy with ns-2, Simulink capabilities have been extended making a simulation outcome more accurate with respect to the component communication. **Figure 20** illustrates the architecture of the PiccSim framework.

In a similar vein, other platforms such as Modelica [26] for modeling and simulating physical systems and ADEVS [27], which is a general purpose discrete-event simulator, also bind with ns-2 and build on the idea of co-simulation.

Veins

Targeting a particular kind of systems and reusing the concept of co-simulation, the Veins framework [28] provides a facility for simulating vehicular area networks (VANETs). For that purpose, it combines two well-established simulators: OMNeT++ [29] and SUMO [30]. The former one is an extensible, modular network simulator coming with different libraries extending its simulation possibilities. An example is MiXiM [31] – a physical layer modeling toolkit equipped with low-level wireless communication protocols (e.g. ZigBee).

SUMO, on the other hand, is a road traffic simulator designed to simulate large-scale vehicular mobility scenarios. It is highly customizable and allows for creation of

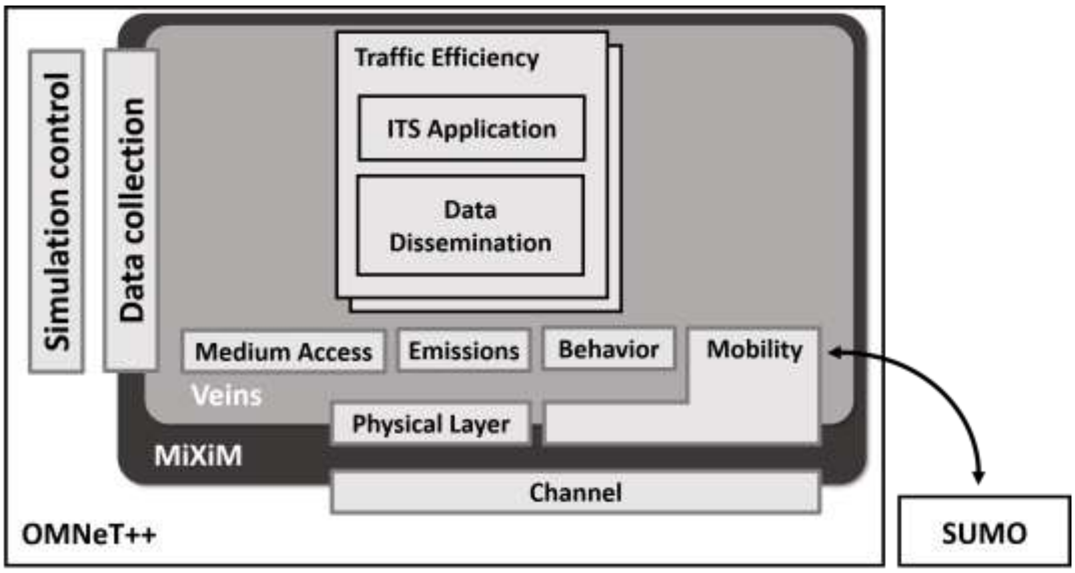


Figure 21. Veins architecture.

different traffic profiles over real geographical locations (support for OpenStreetMap maps [32]).

Veins provides for bi-directional binding between the two simulators, where vehicular network influences the road traffic and vice versa. This way complex vehicle-2-vehicle (V2V) and vehicle-2-infrastructure (V2I) interactions can be modeled and analyzed.

The general architecture of the Veins simulator is illustrated on **Figure 21**.

2.3.1 Summary

In the context of SCPS validation and verification, the co-simulation technique seems to be a promising approach, since it provides a possibility for reusing detailed models for both cyber and physical domains. Moreover, the aforementioned reusability of existing tools and their co-execution solves the problem of complexity that one needs to face when trying to develop from scratch a tool support for SCPS verification. In addition, the ampleness of possible SCPS scenarios requires a considerable flexibility in terms of simulated features. Specifically, if one is to consider use-cases from areas that differ in terms of physical characteristics of the deployment environment - such as VANETs and home automation, where the former heavily relies on component mobility while the other not so much. In that sense, co-simulations facilitate creation of a tool that would support feature enabling depending on requirements of a simulated scenario.

In general, however, existing co-simulation tools do not provide appropriate abstractions for self-organizing architectures. This comes as one of the main limitations when considering application of any of those in the context of SCPS verification.

Chapter 3

Overview of the Contribution

3.1 DEECo Framework

The results of the author's research work around SCPS contribute to the overall development of the DEECo (Dependable Emergent Ensembles of Components) framework. The main purpose of DEECo is to deliver a set of tools that would facilitate different tasks performed by a designer and developer at each of the stages of the SCPS lifecycle. As such, the framework proposes the DEECo component model delivering appropriate abstractions that allow for reasoning about SCPS constituents and their dynamic interactions yet at the design phase of the system. Furthermore, the jDEECoSim platform enables simulation-based techniques for system validation and verification. Finally, the jDEECo runtime environment provides for deployment and execution of a developed SCPS. In the remainder of this section, brief descriptions are given for each of those parts of the DEECo framework. More detailed descriptions follow in Section 4, being incorporated across the collection of published papers.

3.1.1 DEECo component model

In its very basics, the DEECo component model [33] introduces two first class constructs for modeling stakeholders of an SCPS scenario and for expressing dynamic interactions between them. For that, DEECo uses the concepts of component and ensemble (introduced by EBSE – see Section 2.1.3), which correspond to the SCPS constituents and their ad-hocly formed groups of communication.

Components in DEECo are autonomous units of computation and deployment. They consist of a state, which is expressed by a set of attributes (also referred as knowledge) and a functionality defined by processes that can be triggered either in a periodic manner or in reaction to a change in component knowledge. Processes update the state of the component and as such their both input and output are particular attributes from component knowledge. Knowledge of a component is accessed via interfaces, enabling restrictions allowing a component to expose only some parts of its state. Both component knowledge and its functionality are extensible, which improves the overall reusability.

To handle components interactions, DEECo builds over the idea of an ensemble, seen in DEECo as a first-class concept. In a nutshell, DEECo ensemble is a binary relation defined over a state of two components, where one takes the role of the ensemble *coordinator* and another takes the role of its *member*. It consists of a logical predicate called *membership condition* formulated using the attributes from both the coordinator's and the member's knowledge. A positive evaluation of the ensemble membership condition (i.e. its logical predicate is satisfied) triggers the execution of the second part of the ensemble specification, which corresponds to the knowledge exchange between the coordinator and member. DEECo assumes ensembles being the only form of inter-component communication description and realization. As such components exchange data implicitly (similarly to HELENA – see Section 2.1.3.1), which allows system designers to reason about each component independently. This simplifies not only system design phase but also the component development process. **Figure 22** illustrates an example of an SCPS modeled with the use of DEECo components and ensembles.

The DEECo component model brings all the advantages of the architectures described in Section 2.1. If we take into account CBSE and its main characteristics (described in more details in Section 2.1.1), components in DEECo semantics also allow for separation of concerns, which in turn improves the development of components as they can be engineered independently and in isolation. As already mentioned, each component in DEECo is extensible and reusable making them deployable in different contexts.

Furthermore, looking at DEECo components from the perspective of AOSD, the DEECo component model, in that respect, builds heavily on the ideas of autonomous

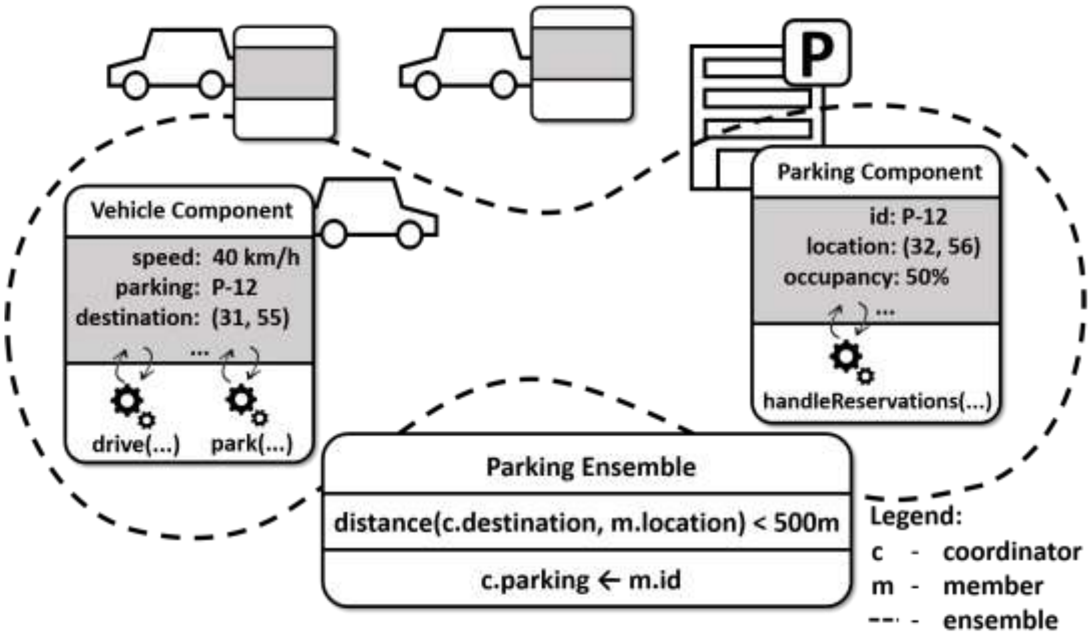


Figure 22. An example of the DEECo design.

agent and applies similar approaches in the context of its components. As a result, components in DEECo can be regarded as agents, which makes the DEECo framework an alternative solution for AOSD development. Unlike agents, however, DEECo components do not use an explicit form of communication. As a matter of fact, components there are unaware of any communication-related aspects, as all of those have been extracted into ensembles. This improves considerably the design process and mitigates the complexity-related issues in AOSD (see Section 2.1.2). Finally, the DEECo component model is a reification of the EBSE, as its main constructs (i.e. component and ensemble) are another interpretation of the ideas of EBSE.

3.1.2 jDEECo

The DEECo framework delivers also the jDEECo runtime environment [34]. It is a distributed Java realization of the DEECo component model that provides for (i) mapping between model-level abstractions and Java language constructs, (ii) automated process execution and ensemble evaluation as well as (iii) component knowledge and network management.

Tuple Spaces and their distributed implementations have inspired the way the jDEECo runtime manages component states (see Section 2.2.3). In order to achieve a full distribution of the runtime environment, the jDEECo platform relies on a component state replication technique that is applied by each of the jDEECo deployment unit (i.e. an instance of the jDEECo runtime). Every such a unit communicates the state of its local and remote components (that it is aware of) to the network incorporating the Gossip style (see Section 2.2.1) of data dissemination. By that, components have access to (at least partial) state of the system. This idea is illustrated in **Figure 23**.

The runtime leverages on the layered architecture design where data access layer is provided with simple interface methods allowing only for component data retrieval or

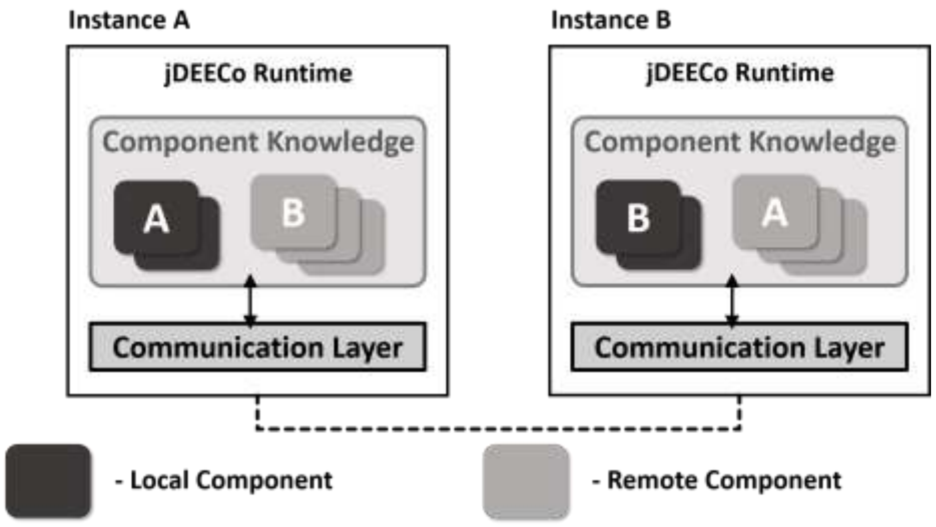


Figure 23. The jDEECo component knowledge replication mechanism

update. This has been decided by taking into account the fact that the whole data management part of the runtime is actually the most complex one, thus, it seemed reasonable to façade it with the LINDA-like operations (see Section 2.2.3) and hide its complexity from the rest of the system. Moreover, in the case of future changes to the jDEECo implementation, its replacement, in such a case, is a straightforward task. As a matter of fact, most recently, the jDEECo platform has been refactored to support custom extensions, allowing jDEECo developers to provide their own solutions for different parts of the platform – including the data management layer.

At the bottom of the jDEECo layered architecture is the communication layer, which builds on the message passing paradigms (see Section 2.2.1), and provides its own specialized solution for inter-component data dissemination that is tailored to support component autonomy and mobility, open-endedness of SCPS, and deployment infrastructure. Like in VANET networks, the jDEECo supports two deployment types: infrastructure-based and infrastructure-less. The prototype implementation is designed to distribute component state information over two different channels: one dedicated for the former network type and other to the latter one. Therefore, the jDEECo platform supports heterogeneous deployments, which comprise a combination of both infrastructure-based and infrastructure-less networks.

As a proof of the concept, the infrastructure-less network has been evaluated (and its support is a part of the jDEECo implementation) on the MAC layer of WSN ZigBee protocol complying with the IEEE 802.14.5 specification [35]. Its selection was based mainly on the characteristics of the protocol (i.e. supported communication range, message size etc.) as well as its popularity in the domain of LoWPAN (Low power Wireless Personal Area Network) [HC08, Mul07]. Resignation from the use of existing higher-level protocols (like the aforementioned ZigBee [36] or Z-Wave [37]) has been justified mainly by the lack of support for channel-level broadcast, where data is propagated only to the most immediate neighbors (i.e. those in the radio range). This allowed for dropping costly routing algorithms (e.g. AODV [PBRD03]) in favor of a proprietary solution tailored to component state replication and dissemination.

Finally, the upper-level message exchange algorithm implements the Gossip protocol (see Section 2.2.1), in order to deal with the opportunistic nature of the MANETs. It is implemented in two versions supporting both deployment types (i.e. infrastructure-based and infrastructure-less).

On top of those, two optimization techniques have been introduced to the whole process of the component data exchange (see Sections 4.3 and 4.5). Their main goal is to reduce the amount of data being sent to the network, improving its overall utilization and reducing delays in data passing. The Publish-Subscribe model has been used to implement one of the optimization techniques. In principle, the method reduces the amount of messages produced by the Gossip protocol in the infrastructure-based networks. The idea behind is to use broker-like entities to decide centrally on components membership to a particular ensemble or in other words, to say which components are

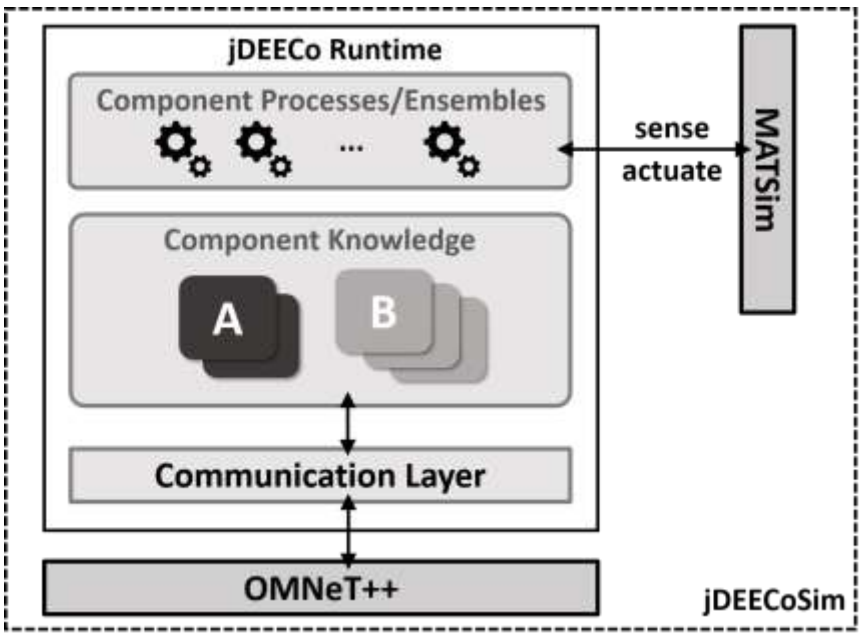


Figure 24. jDEECoSim architecture overview

supposed to communicate with each other. It is important to note, that the entire technique is optional and is not crucial for system operability. More details on this particular idea can be found in Section 4.5.

3.1.3 jDEECoSim

The last part of the author's contribution and at the same time one of the tools provided by the DEECo framework is the simulation framework called jDEECoSim [34]. To this date, it integrates the jDEECo runtime environment, the OMNeT++ network simulator [29] and the MATSim traffic simulator [38]. All knowledge exchange passed between components is routed through OMNeT++, which is meant to provide close-to-real network latency estimates w.r.t. to network topology, the geographical position of components, network collisions, and packet drops, etc. Moreover, by including the INET [39] and MiXiM [31] – OMNeT++ extensions, jDEECoSim allows for simulating deployments in a mixed network environment combining infrastructure-based and infrastructure-less networks. MATSim, on the other hand, facilitates mobility simulation and allows for modeling large-scale traffic scenarios. Accessing appropriate information (e.g. current position, trajectory) is provided to DEECo components by a convenient concept of sensors and actuators.

In the end, jDEECoSim is a tool that can be applied at the post-development stage to verify SCPS behavior with respect to its specification just before deploying it in real-life settings.

3.2 Research goals revisited

In terms of goal **G1**, the author has provided several proves of concept for the DEECo component model. Its applicability, in the design phase of an SCPS, has been validated on multiple scenarios scattered across author's publications.

The main contribution of the author, however, is the prototype implementation of the jDEECo deployment and execution environment that addresses challenge **C2**, by achieving the goal **G2** (including sub-goals **G2a** and **G2b**). As stated in **G2** (and described in the previous section), the runtime environment automates component state interchange across the network, building over the gossip protocol. Regarding the sub-goal **G2a**, the runtime environment supports heterogeneous deployment over both infrastructure-based and infrastructure-less networks, exemplified by utilizing the IP network-layer protocol and 802.15.4 link-layer protocol for LoWPAN respectively.

To satisfy the sub-goal **G2b** the environment implements two optimization mechanisms, specialized with respect to both DEECo component model and supported network types (i.e. infrastructure-based and infrastructure-less). The optimization techniques concentrate on reducing the amount of data propagated over the network, which results in better utilization of the communication medium (see Sections 4.3 and 4.5).

Regarding the goal **G3**, the runtime environment has been integrated with the simulation tool jDEECoSim (contributed also by the author and being a part of the jDEECo framework) that allows for experimentations during the final stages of the SCPS development process. By integrating together two well-known simulation frameworks, the platform is capable of simulating simultaneously two aspects of the developed system: the network infrastructure and mobility of components, which combined are meant to resemble the real-life deployment. jDEECoSim itself does not provide any guarantees over the critical aspects satisfiability of a developed SCPS. However, it can support any formal analyzes by delivering their validation through an experimental evidence. In general, the platform is designed to facilitate the system behavior verification process and assessment of its (non-critical) requirements satisfiability.

The runtime environment and the simulation platform have already been evaluated on multiple case studies that stemmed from different projects involvement, where some of them were conducted under the umbrella of the EU FP7 Programs (i.e. ASCENS and RELATE) and some were part of a bilateral cooperation with partner institutions (e.g. Volkswagen AG, Chemnitz University). Moreover, both served also as a proof of concept for other ideas (e.g. a design technique for DEECo-based systems – Invariant Refinement Method [KBP+13], and inaccuracy analyses for component belief [AABG+14c]), researched separately but also in the context of the DEECo component model.

3.3 Selected Publications

The following list of publications consists of items that are considered to consist of a major author's contribution reflected mainly in their evaluation parts.

The first two articles introduce the DEECo component model together with its core abstractions – *component* and *ensemble*, addressing goal **G1**. In the second publication, the jDEECo runtime is introduced (partially addressing goal **G2**), which has been solely prototyped by the author.

- [KBPK12] Keznikl J., Bureš T., Plášil F., Kit M.: "Towards Dependable Emergent Ensembles of Components: The DEECo Component Model", In Proceedings of WICSA/ECSA 2012, Helsinki, Finland, August 2012

- [BGH+13] Bureš T., Gerostathopoulos I., Hnětynka P., Keznikl J., Kit M., Plášil F.: "DEECo - an Ensemble-Based Component System", In Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE 2013), Vancouver, Canada, June 2013

Furthermore, the next publication introduces an extension to the DEECo component model by the concept of *communication boundary* that allows for communication optimization when it comes to system realization over an infrastructure-less networks. Specifically, the author has contributed in the evaluation part of the article, which is the first time when a simulation technique has been used in the context of the DEECo component model and as such setting the foundations for the jDEECoSim simulation platform. In particular, the work presented in this publication addresses the goal **G2** including both **G2a** and **G2b**.

- [BGH+14b] Bureš T., Gerostathopoulos I., Hnětynka P., Keznikl J., Kit M., Plášil F.: "Gossiping Components for Cyber-Physical Systems", In Proceedings of the 8th European Conference on Software Architecture (ECSA 2014), Best Research Paper Award, Vienna, Austria, August 2014

The next item, on the publication list, is a technical report that formally specifies the DEECo computational model. It has been formulated along with the [BGH+14b] and serves as a formal description of DEECo and starting point for any further extensions (modifications) introduced to the model. It addresses the semantic part of the goal **G1**.

- [BGH+14a] Bureš T., Gerostathopoulos I., Hnětynka P., Keznikl J., Kit M., Plášil F.: "Computational Model for Gossiping Components in Cyber-Physical Systems", Tech. Report No. D3S-TR-2014-03, Dep. of Distributed and Dependable Systems, Charles University in Prague, April 2014

To argument on addressing of goals **G2a** and **G2b**, the following work introduces an extension to the DEECo component model that in a similar vein as [BGH+14b] strives to optimize component state dissemination across the deployment network. This time, however, the focus is centered around the infrastructure-based networks.

- [KPM+15] Kit M., Plášil F., Matěna V., Bureš T., Kováč O.: "Employing Domain Knowledge for Optimizing Component Communication", In Proceedings of the 18th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE 2015), Montreal, Canada, May 2015

The next publication is a tool presentation, introducing the DEECo framework as a solution supporting the implementation of self-adaptive techniques in the context of general CPS development. It also introduces the jDEECoSim platform as a verification tool using the co-simulation method by combining the two well-known simulators OMNeT++ and MATSim (see Section 3.1.3). This article arguments addressing of both **G2** and **G3**.

- [KGB+15] Kit M., Gerostathopoulos I., Bureš T., Hnětynka P., Plášil F.: An Architecture Framework for Experimentations with Self-Adaptive Cyber-Physical Systems, In Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015), Florence, Italy, May 2015

As the final item on the list of the author's major contributions, is the following publication being a result of a collaborative research work between the author's research group and the group from the Chemnitz University. This work proposes a DEECo proprietary method for analyzing an SCPS system with respect to its critical aspects – i.e. timely aspects specified in its non-functional requirements. Similarly, the author contributed mainly in the evaluation part, providing for simulation-based experiments for the method, which addresses goal **G3** from Section 1.3**.**

- [MKM+16] Masrur A., Kit M., Matěna V., Bureš T., Hardt W.: "Component-Based Design of Cyber-Physical Applications with Safety-Critical Requirements", To appear in Microprocessors and Microsystems, Elsevier, 2016

Furthermore, the following list of publications is considered to have minor author's contribution:

- [BDNG+13] Bureš T., Rocco de Nicola, Gerostathopoulos I., Nicklas Hoch, Kit M., Nora Koch, Giacoma Valentina Monreale, Ugo Montanari, Rosario Pugliese, Nikola Serbedzija, Martin Wirsing, Franco Zambonelli: "A Life Cycle for the Development of Autonomic Systems: The e-Mobility Showcase", In Proceedings of the 3rd Workshop on Challenges for Achieving Self-Awareness in Autonomic Systems, Philadelphia, USA, IEEE, September 2013

- [AAGGH+14] Al Ali R., Gerostathopoulos I., Gonzalez-Herrera I., Juan-Verdejo A., Kit M., Surajbali B.: "An Architecture-Based Approach for Compute-Intensive Pervasive Systems in Dynamic Environments", In Proceedings of International Workshop on Hot Topics in Cloud service Scalability, ICPE '14, Dublin, Ireland, 2014

- [GKB+14] Gerostathopoulos I., Keznikl J., Bureš T., Kit M., Plášil F.: "Software Engineering for Software-Intensive Cyber-Physical Systems", Presentation at the CPSData Workshop: Big Data Technologies for the Analysis and Control of complex Cyber-Physical Systems, September 2014

- [AABG+14b] Al Ali R., Bureš T., Gerostathopoulos I., Hnětynka P., Keznikl J., Kit M., Plášil F.: "DEECo: an Ecosystem for Cyber-Physical Systems", In companion proceedings of the 36th International Conference on Software Engineering (ICSE 2014), Hyderabad, India, ACM, poster and extended abstract, June 2014

- [MKBH14] Masrur A., Kit M., Matěna V., Bureš T., Hardt W.: "Towards Component-Based Design of Safety-Critical Cyber-Physical Applications", In Proceedings of the 17th Euromicro Conference on Digital Systems Design (DSD 2014), Verona, Italy, August 2014

- [BHK+14] Bureš T., Horký V., Kit M., Marek L., Tůma P.: "Towards Performance-Aware Engineering of Autonomic Component Ensembles", In Proceedings of the 8th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014), Corfu, Greece, Springer, October 2014

- [BGH+15] Bureš T., Gerostathopoulos I., Hnětynka P., Keznikl J., Kit M., Plášil F.: "The Invariant Refinement Method, Software Engineering for Collective Autonomic Systems", Results from the ASCENS Project, Springer-Verlag, in press, 2015

Collection of Papers

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

Jaroslav Keznikl,
Tomáš Bureš,
František Plášil,
Michał Kit

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

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

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

Summary of the Paper

This work is a first attempt to introduce the idea behind components ensembles delivered by the DEECo component model and its abstractions. It relies on a simplistic case study of autonomous robots coordinating between each other on a crossroad and deciding on the order in which robots cross the intersection. The problem is fully distributed, which means that there is no centralized entity that the robots could refer their decisions. They need to rely purely on the exchanged information and make decisions based on that. The scenario and its requirements are drawn in sections II and III accordingly. Furthermore, in Section IV, the scenario is modeled using the DEECo constructs – i.e. components and ensembles – expressed in the dedicated DSL. In addition, a brief explanation on the communication model is given, however, it is not yet backed by any proof of the concept. This is highlighted in Section V, which consists the envisioned plan for the future and challenges that need to be addressed.

Author Contribution and Goals Addressed

The author's contribution to this paper was participation in the formulation of the DEECo component model and its main features. Although, this work does not consist any realization proving the proposed ideas, it has been prototyped and validated by the author. However, due to the immaturity of the solution, it has not been included in the final version of the paper.

In reference to the research goals defined in Section 1.3 of this thesis, the goal **G1** is addressed by the work published in this article. By delivering the DEECo component model and its abstractions (i.e. components and ensembles), the SCPS characteristics specified in the goal definition (i.e. component autonomy and architectural dynamicity) are considered to be satisfied.

2012 Joint Working Conference on Software Architecture & 6th European Conference on Software Architecture

Towards Dependable Emergent Ensembles of Components: The DEECo Component Model

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

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

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

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

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

I. Introduction

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

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

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

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

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

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

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

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

II. Case Study

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

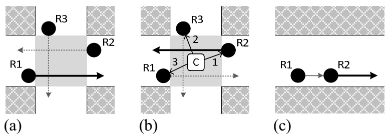


Figure 1. Robot Case Study: (a) autonomous robots, (b) robots advised by a crossing, (c) convoy

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

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

III. Overview of Requirements

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

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

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

IV. DEECo Component Model

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

A. Component Structure

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

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

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

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

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

Figure 2. Example of a DEECo component

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

Figure 3. Example of an ensemble prescription

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

B. Component Composition

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

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

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

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

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

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

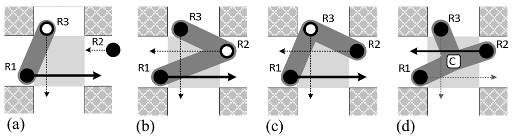


Figure 4. Ensemble Examples: (a) two-robot ensemble with coordinator R3, (b) autonomous robots ensemble with coordinator R2, (c) autonomous robots ensemble with coordinator R3, (d) crossing ensemble

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

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

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

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

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

C. Computational Model

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

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

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

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

V. Discussion: Vision and Challenges

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

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

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

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

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.2 DEECo: an Ensemble-Based Component System

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

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

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

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

Summary of the Paper

The aim of the following publication is to introduce the DEECo component model as a solution for designing and building architecturally dynamic systems (which is the case of SCPS) built over the idea of component ensembles. The core concepts of the DEECo component model (i.e. components and ensembles) are illustrated on the e-mobility case study, being one of the three key example scenarios of the ASCENS Project. The scenario consists of vehicles and parking lots as the stakeholders. Vehicles coordinate with parking lots in order to reserve parking space. The selection of the parking lot is based on its proximity to the vehicle's point of interest and the occupancy of the parking lot. The full description of the scenario can be found in Section 2 of the paper. The scenario serves as an example for introducing the DEECo component model abstractions detailed in Section 3. Apart from the descriptive presentation of the abstract features of the model, the publication delivers a prototype implementation of the jDEECo runtime environment as well as the first implementation of the e-mobility scenario – Section 4. Further, in Section 5, the IRM (Invariant Refinement Method) [KBP+13] is described and exemplified on the use-case scenario. The IRM method is researched separately to the work presented in this thesis. Nevertheless, being a part of the DEECo framework, these two topics are complementary and strongly related, even though, they address different stages of the software engineering process. While jDEECo and jDEECoSim focus on the development and deployment phases, the IRM addresses the design and modeling part of the process. Section 6 gives a short discussion on the observations and experience gained during the work on the implementation of the e-mobility example using the jDEECo approach. Finally, the state of the art is given in Section 7, which is then followed by conclusions and future work.

Author Contribution and Goals Addressed

The research goals addressed in this work are goals **G1** and **G2**. They talk about proper modeling abstraction for SCPS development as well as the demand for deployment platform that would support automation in SCPS component management. Specifically, Section 3 delivers the DEECo component model as the means for modeling ensemble-based systems, which are in the context of this thesis an alias for SCPS. Moreover, taking into account, the very first prototype implementation of the jDEECo runtime given in Section 4 of the paper, also goal **G2** is addressed at least partially.

The author's contribution to this work consists of participation in the DEECo component model formulation as well as the implementation of the prototype for the jDEECo platform together with the realization of the e-mobility case study.

The e-mobility case study presented in this paper has been continued (later after this work has been published) in the form of multilateral agreement between the author's research group, German research institute – Fraunhofer FOKUS [40] and the Volkswagen [41] research group. It has been extended with different aspects including, the Volkswagen proprietary (hence a reference is not available) route planning utility designed for next generation electric vehicles as well as global and local level optimizations with respect to vehicle route planning and parking lot occupancy. This, cooperation with the industrial partner brought precious experience that was beneficial for further research on the DEECo framework. Specifically, in the context of the thesis, this refers to the component model, jDEECo runtime environment, and jDEECoSim platform.

DEECo – an Ensemble-Based Component System

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

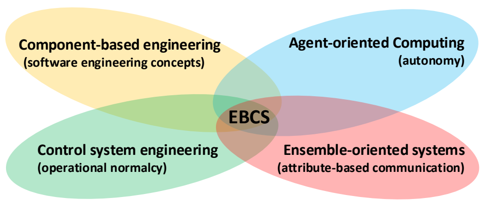

Component-based engineering
(software engineering concepts)
Agent-oriented Computing
(autonomy)
EBCS
Control system engineering
(operational normalcy)
Ensemble-oriented systems
(attribute-based communication)

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

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

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

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

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

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

1.2 Goals and Structure of the Text

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

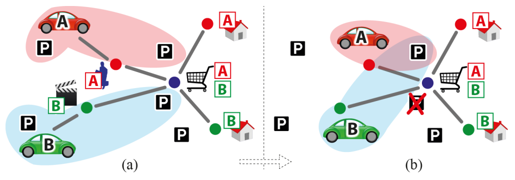


Figure 2: E-mobility: Potential ensembles and their dynamic changes (available parking stations close to respective POIs).

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

2. RUNNING EXAMPLE

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

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

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

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

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

3. DEECo COMPONENT MODEL

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

3.1 General Concepts

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

3.1.1 Component

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

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

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

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

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

Figure 3: Examples of DEECo component definitions in a DSL.

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

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

3.1.2 Ensemble

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

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

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

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

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

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

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

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

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

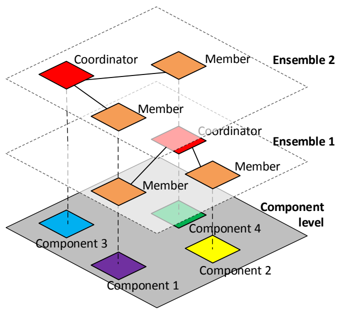


Figure 5: Composition of components into multiple overlapping ensembles in DEECo.

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

3.2 Computational Model

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

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

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

4. DEECo REALIZATION IN JAVA

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

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

4.1 Mapping of DEECo Concepts to Java

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

4.1.1 Component

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

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

Figure 6: Example of a component definition in Java.

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

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

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

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

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

Figure 7: Example of an ensemble definition in Java.

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

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

4.1.2 Ensemble

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

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

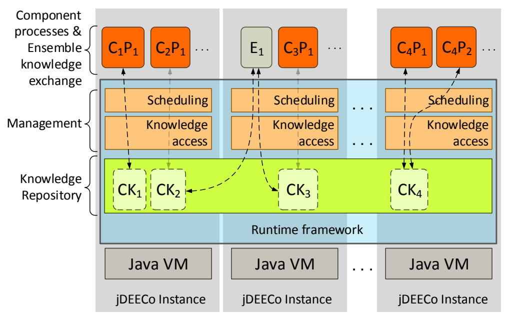


Figure 8: jDEECo runtime framework architecture.

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

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

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

4.2 Runtime framework

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

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

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

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

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

4.3 Tool support

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

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

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

5. SOFTWARE ENGINEERING PROCESS INTEGRATION

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

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

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

5.1 Basic Concepts of IRM

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

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

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

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

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

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

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

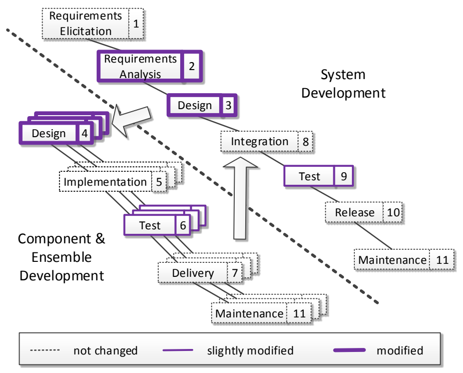


Figure 9: Example of IRM integration into the reference CBD process of [12].

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

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

5.2 Integration with CBD Process

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

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

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

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

Table 1: IRM injection points into the CBD process.

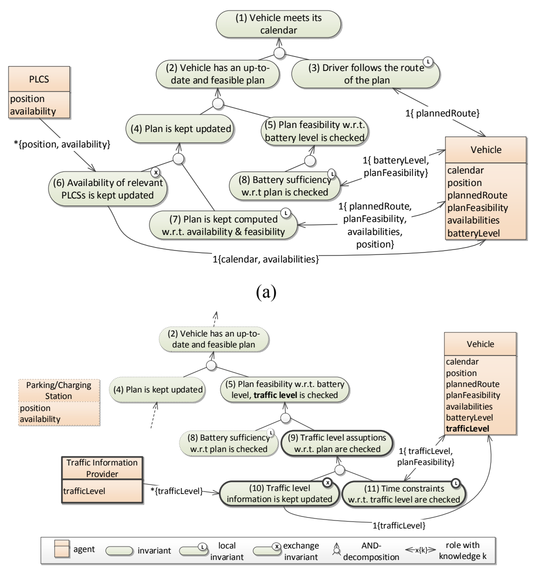
5.3 System Evolution

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

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

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

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


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

(b)

Figure 10: Capturing system evolution in IRM.

6. EXPERIENCE

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

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

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

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

7. RELATED WORK

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

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

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

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

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

8. CONCLUSIONS AND FUTURE WORK

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

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

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

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

9. ACKNOWLEDGMENTS

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

10. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.3 Gossiping Components for Cyber-Physical Systems

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

In proceedings of the **8th European Conference on Software Architecture (ECSA 2014)**.

Awarded the Distinguished Paper Award.

Published by Springer,
pages 250-266,
ISBN 978-3-319-09969-9,
August 2014.

The original version is available electronically from the publisher's site at http://dx.doi.org/10.1007/978-3-319-09970-5_23.

Summary of the Paper

The paper introduces a gossip-based communication protocol for the DEECo component model realization (i.e. jDEECo) extended by the idea of *communication boundary* that enriches the ensemble specification. The main principle, behind it, is to prevent the data propagation mechanism from polluting the underlying network with irrelevant (with respect to the application logic) data. It is important, especially, if the shared communication medium (such as a radio channel in case of wireless networks), is taken into account. The concept of the communication boundary has been validated on the firefighters use-case scenario that involves multiple rescue teams working in separate (but geographically close) locations – see Section 2 of the paper. The scenario main components (i.e. firefighters) have been expressed using DEECo abstractions and the main challenges of the approach have been identified there. Further, in Section 3, the gossip-based communication model is presented together with the distributed ensemble evaluation method. Finally, the core idea of the communication boundary is given and proposed as an extension to the DEECo ensemble specification. Then, based on the firefighters scenario the proposed techniques have been evaluated using the prototype implementation of the jDEECoSim framework (the name was not explicitly used there), consisting the integration of the jDEECo runtime only with OMNeT++ (Sections 4 and 5). Section 6 gives an elaboration on the main challenges and open questions related to the proposed solution. The paper is then continued with the description of selected representatives of the state of the art (Section 7) and concluded with Section 8, drawing a plan for the future work.

Author Contribution and Goals Addressed

The work presented in this paper addresses the goal **G2a** as it targets the infrastructure-less deployment of the jDEECo platform. In addition, the goal **G2b** is addressed by the introduction and implementation of the concept of communication boundary that optimizes considerably the underlying network utilization.

The author’s contribution in the context of this paper was the participation in the formulation and implementation of both the gossip-based communication protocol and boundary condition into the jDEECo platform.

The paper has been awarded the best paper award at the ECSA'14 venue.

Gossiping Components for Cyber-Physical Systems

Tomas Bures[1,2], Ilias Gerostathopoulos[1], Petr Hnetynka[1], Jaroslav Keznikl[1,2], Michal Kit[1], and Frantisek Plasil[1]

[1]Faculty of Mathematics and Physics, Charles University in Prague, Prague, Czech Republic
[2]Institute of Computer Science, Academy of Sciences of the Czech Republic, Prague, Czech Republic
{bures, iliasg, hnetynka, keznikl, kit, plasil} @d3s.mff.cuni.cz

Abstract. Developing software for dynamic cyber-physical systems (CPS) is a complex task. One has to deal with the dynamicity and unreliability of the physical environment where the software resides in, while, at the same time, provide sufficient levels of dependability and scalability. Although emerging software engineering abstractions, such as dynamic ad-hoc component ensembles, provide a convenient way to structure software for dynamic CPS, they need to be mapped to robust decentralized execution schemes in real-life settings. A particular challenge in this context is the robust distributed data dissemination in dynamic networks. Gossip-based communication stands as a promising solution to this challenge. We argue, that exploitation of application-specific information, software architecture in particular, has a large potential for improving the robustness and performance of gossip-based communication. This paper proposes a synergy between high-level architectural models and low-level communication models to effectively enable application-specific gossiping in component-based systems. The synergy is exemplified on the DEECo component model which is tailored to the needs and specifics of CPS, and evaluated on an emergency coordination case study with realistic network configurations.

Keywords: Component, Ensemble, Gossip, Cyber-Physical Systems, MANET.

1 Introduction

Cyber-physical systems (CPS) are complex networked systems where the interplay of software control with the physical environment has a prominent role. Examples range from intelligent navigation systems (cars that communicate with each other and with street infrastructure to minimize traffic congestion, fuel consumption, etc.) to emergency coordination systems. Modern CPS are inherently distributed on a large scale and consist largely of mobile devices. They are also increasingly depending on software which has actually become their most intricate and extensive constituent [1].

Building software for large-scale software-intensive CPS via systematic software engineering approaches is a notoriously difficult task. This stems from the fact that CPS invalidate most of the assumptions that typically hold in software engineering of general-purpose systems [2]. Whereas the challenges and opportunities of CPS cover a range of areas, in this paper we focus on the communication requirements of CPS. In CPS, the physical substratum continuously evolves following the movement of mobile devices. Locality of devices directly affects reachability and connectivity. Communication between devices is opportunistic; there are no guarantees regarding the stability and reliability of the established links. The network topology itself is dynamic and often relies on ad-hoc means without any managing infrastructure. Finally, the environments where CPS operate (e.g., road networks, emergency sites) are highly dynamic and inherently unpredictable.

At the same time, CPS have also a number of specifics that can be advantageously exploited, such as the fact that by moving around in the environment, the wireless devices effectively enlarge the physical area where information can be disseminated [3]. Physical locality and location-dependency of data offer also a natural way to partition the system and provide built-in scalability and robustness.

Looking at the state-of-the-art in distributed communication, gossip and epidemic protocols provide an efficient way to address the aforementioned specifics. Gossip protocols cope with node and network failures, are scalable due to their symmetric nature, and can exploit the physical mobility of gossiping nodes [3]. The gossiping paradigm has already been applied with success in both Internet-based systems and wireless mobile ad-hoc networks (MANETs) [4].

The central idea in gossip protocols is the periodic and probabilistic data transmission from a source node to a set of selected peers [4–6]. They typically combine probabilistic forwarding with counter-based, distance-based, and location-based mechanisms. These mechanisms and configuration parameters are, however, only available at the lower level of the software stack, often transparent to the application/architecture layer. While this is reasonable for uniform data dissemination, it becomes problematic when the spread of data depends on the architectural configuration in question.

The problem lies in a significant abstraction gap between gossip protocols and application-level architecture design using component models tailored to CPS.

In this paper, we aim at bridging this gap by incorporating concerns of gossiping into sound software engineering abstractions, which allow for (i) systematic engineering of CPS via gossiping components and (ii) application-specific, scalable, and efficient gossip-based communication. We do so in the context of DEECo [7] – a component model that specifically targets dynamic, ever-changing architectures of CPS by relying on the concepts of autonomous (soft) real-time components, and dynamic ad-hoc component ensembles. Our approach is not limited to DEECo though, since it is based on the generic synergy between a set of high-level architectural abstractions supporting dynamicity and low-level primitives of gossip-based protocols.

The rest of the text is structured as follows. In Section 2, we elaborate on a scenario from an emergency coordination case study that provides the motivation for architecture-based decentralized solution. Section 3 presents our approach and its integration into DEECo, while Section 4 outlines the implementation. Following, Section 5 presents the simulation-based evaluation results. Section 6 discusses key contributions and emerging related challenges. Finally, in Section 7 we survey the related work and in Section 8 we present our conclusions.

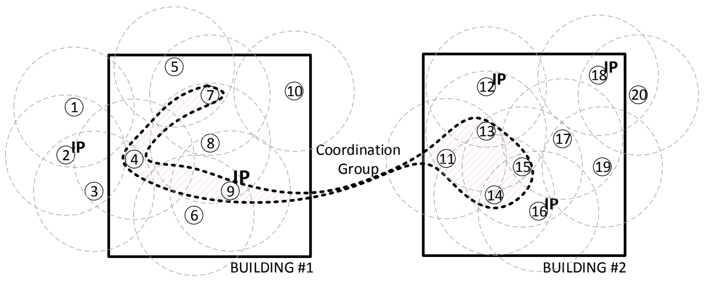


Fig. 1. Motivating scenario: Mobile and stationary nodes cooperate via ad-hoc coordination groups that span within designated boundaries.

2 Motivating Scenario

To illustrate the need for effective mapping of architecture-level concepts to decentralized communication schemes in CPS, we use a scenario taken from a firefighter coordination case study[1], which is a real-world real-scale case study for evaluating distributed adaptive systems.

In the scenario, firefighters belong to tactical groups corresponding to the mission in hand. In case of an emergency, a scouting team composed of a team leader and several team members is initially dispatched to the operation site with the goal to assess the criticality level of the situation in hand, so that appropriate strategic decisions can be taken (e.g., mission escalation, request for additional teams). A strong requirement for the effective cooperation of team members is efficient data dissemination – every member has to be notified in a timely manner about important events and threats (e.g., low oxygen level in a particular room, firefighter in danger because of high temperature level) so that the team can act collaboratively and proactively.

Firefighters are equipped with low-power devices with sensing and actuating capabilities that are integrated into their personal protection equipment (being thus mobile). The devices communicate primarily via wireless mobile ad-hoc network (MANET) protocols (e.g., IEEE 802.15.4); additionally, some devices have IP connectivity. Advantageously, the firefighters may exploit other devices on the fire scene (e.g., on-site access points or devices of other emergency personnel) as network relays to boost their wireless coverage and performance. For illustration, consider an operation site that consists of two buildings (Fig. 1).

Obviously, the key challenges stem from the dynamicity of the whole scenario; in particular, the issues to be addressed include (i) MANET management and efficient use of the communication medium and (ii) seamless inclusion of the related concepts in the high abstraction level employed in the design of the corresponding software architecture.

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

```
1.  role TemperatureSensor:
2.    missionID, temperature
3.
4.  role TemperatureAggregator:
5.    missionID, firefightersInDanger, temperatures
6.
7.  component Firefighter13 features TemperatureSensor:
8.     knowledge:
9.        ID = 13, missionID = 1024, position = {50.075306, 14.426948}, oxygenLevel = 90%, temperature = 35.2
10.    process measureTemperature (out temperature):
11.          temperature ← Sensor.read()
12.       scheduling: periodic( 1000ms )
13.    … /* other process definitions */
14. … /* other firefighter definitions */
15.
16. component Leader features TemperatureAggregator:
17.    knowledge:
18.       ID = 2, missionID = 1024, position = {50.075310, 14.426952}, firefightersInDanger = {1,3, …},
19.       temperatures = {{1,30.7}, {2,25.0}, {3,35.2},…}
20.    process findFirefightersInDanger(in temperatures, out firefightersInDanger):
21.          firefightersInDanger ← analyze(temperatures)
22.       scheduling: periodic( 500ms )
23.    … /* other process definitions */
24.
25. ensemble TemperatureUpdate:
26.    coordinator: TemperatureAggregator
27.    member: TemperatureSensor
28.    membership:
29.       member.missionID == coordinator.missionID
30.    knowledge exchange:
31.       coordinator.temperatures ← {  (m.ID, m.temperature) | m ∈ members }
32.    scheduling: periodic( 500ms )
```

Fig. 2. Examples of DEECo components and ensembles of the firefighter coordination case study.

2.1 A DEECo-Based Solution

A promising approach for developing software of dynamic CPS is to employ the DEECo component model and its related methods and tools [7].

The design process in DEECo starts with identifying the main system components and dynamic ad-hoc coordination groups – *ensembles* – that the components should establish in order to cooperate for a common goal. In the scenario, ensembles reflect the groups of firefighters exchanging measured data (e.g., temperature, oxygen level) and the groups of officers exchanging strategic information (e.g., mission updates, orders from the chief officer). For illustration, consider the ensemble definition in Fig. 2, lines 25-32. Here, the goal is to enable the members of a firefighting team to propagate information on the measured temperature to the leader of the team so that the leader can determine which firefighters are in danger. In general, an ensemble definition in DEECo contains a condition specifying which components should be considered for membership (lines 28-29), and a function that specifies knowledge exchange among the members (lines 30-31). A particular ensemble (i.e., an instance of an ensemble definition) is identified by its coordinator which features a specific role (line 26). It is instantiated and dissolved by the DEECo runtime environment (Runtime further on), which periodically (line 32) checks the membership of potential groups of coordinator-members. Within an established ensemble, Runtime periodically performs the knowledge exchange, which transfers data between the coordinator and members.

A component in DEECo is an independent unit of computation and deployment. In the scenario, components correspond to the actors of the system (active firefighter, officer, relay node, etc.). For illustration, consider the two components in Fig. 2. Their state is captured by knowledge (lines 8-9, 17-19) and functionality by processes (lines 10-12, 20-22). Every component features a number of roles, i.e., sets of knowledge fields (lines 1-2, 4-5), which are used as the contract between the component and ensembles. Processes are executed by Runtime in a time- or event-triggered fashion (lines 12, 22). Each process execution consists of atomically reading (a part of) the knowledge of the component, executing the process body, and atomically updating the knowledge with the result.

Note that components in DEECo do not explicitly communicate with each other; their only means of communication is knowledge exchange mediated by the ensembles to which the components belong. A component may belong to a number of ensembles at a time (i.e., ensemble instances may overlap).

2.2 Challenges in DEECo-Based Solution

As shown above, DEECo provides a comprehensive set of concepts at a high level of abstraction, coping with the dynamicity by means of component roles and ensembles. However, mapping the concepts into a scalable and robust DEECo implementation is challenging. The particular challenge lies in how and where to evaluate the membership condition for every possible ensemble. This typically requires reasoning at the system level, exploiting some form of global view over the system state. If this reasoning is encapsulated into a special-purpose entity in Runtime, this entity becomes a bottleneck – single point of failure. In particular, such a centralized solution does not scale when ensembles are to be formed among large numbers of components.

3 Gossiping in Ensembles

In order to mitigate the above issue, we have adopted a fully decentralized and robust approach relying on gossiping for establishing ensembles and performing knowledge exchange. In principle, we replace the network communication layer of DEECo by gossip-based communication and extend the DEECo architectural model (the definition of ensembles in particular) by the concept of a *communication boundary* so as to allow efficient functioning of the underlying gossiping mechanism.

To connect components at the architectural level with their physical deployment, we define *node* as a hardware/software platform where a number of DEECo components are deployed (hosted in an instance of Runtime). Nodes communicate with each other via their network interfaces depending on the available networking infrastructure. Thus, component communication is constrained by the available networking infrastructure between the nodes the components are deployed on. Inspired by the motivating scenario, we focus on combinations of IP-based networks (wireless and wired) and MANET networks (which allow only for short range broadcast communication). As a product of distributed communication among nodes, each node obtains copies – *replicas* – of the knowledge of components hosted on (some of) the other nodes.

The main principles of our approach to gossip-based ensemble creation and knowledge exchange can be characterized by the following points:

1. A node has its own awareness of ensemble instances existing in the system, specifically of those that include the components deployed on the node. This awareness is based on evaluating the membership with respect to the current knowledge of local components and replicas of other components.
2. Based on the awareness obtained in (1), a node performs only knowledge exchange that results in updating the knowledge of the local components using, again, the current knowledge of the local components and replicas of others.
3. A node proactively disseminates component knowledge, so that every other node has the replicas necessary for realization of (1) and (2).

The following describes the individual elements of our approach in more detail – points 1 and 2 are explained in Section 3.1, while point 3 is elaborated in Sections 3.2 and 3.3.

3.1 Decentralized Evaluation of Ensemble Membership/Knowledge Exchange

Instead of forming ensembles by looking at a snapshot of the whole system (which would imply that a global view on the system has to be available), we take a node-centric approach. Every node periodically iterates over all known ensemble definitions and checks whether a local component can act as a member or coordinator in an instance of the ensemble definition, given its replicas. For each such ensemble instance, it performs the corresponding knowledge exchange, which results in updating the local components' knowledge (but not the replicas).

As an example, consider an instance of the TemperatureUpdate ensemble (Fig. 2) evaluated on the site of the coordinator. In this case, the knowledge exchange results into updating the coordinator's field temperatures.

Note that a consequence of this technique is that degradation of system performance when no connectivity is available (e.g., due to appearance/disappearance/mobility of nodes) is gradual: each Runtime effectively operates on the locally available replicas until they become too outdated to rely upon. Here, we count on one of the specifics of CPS, namely on the fact that the values of most magnitudes in CPS (e.g., temperature in Fig. 2) evolve gradually according to physical laws [8]. Practically this means that a belief which is not too old may still be at least partly relevant. Another consequence is that, due to belief outdatedness causing belief inaccuracy, it is possible for a component to behave as if it were in ensemble with a coordinator, which is not aware of it (and vice-versa). These consequences are further analyzed in Section 6.2.

3.2 Asynchronous Knowledge Dissemination via Gossip

The decentralized solution presented in Section 3.1, requires that each node possesses all the necessary replicas from the components that can potentially participate in ensembles with its local components. We enable this by asynchronous gossip-based knowledge dissemination between all the components of a DEECo application.

The main idea is that every node periodically publishes the knowledge of its local components on the network. For MANETs, this translates to periodic broadcast within the wireless range of the node. For IP networks, it translates to periodic sending to randomly selected nodes. Upon reception of a component's knowledge, a node probabilistically decides whether to retransmit the received knowledge. The nodes that perform such re-transmission then act as relays. Here, we rely on the probabilistic convergence of gossip protocols [9], which ensures that every node will eventually receive the knowledge of every component in a bounded number of steps. The nodes that dynamically appear in the system join the publication and re-transmission of knowledge automatically.

Note that this dissemination scheme dictates that all nodes potentially perform the retransmission, not only the ones that are interested in the disseminated knowledge (i.e., nodes hosting components that could be members of the ensemble which the disseminated knowledge relates to).

3.3 Bounding the Gossip

Although the aforementioned gossip-based knowledge dissemination successfully propagates the knowledge of all nodes to all nodes, it raises performance issues. Specifically, if a DEECo application is considered as a ubiquitous ecosystem in a real environment, the application is potentially boundless w.r.t. network reachability. In such a system, unlimited gossiping is not a viable option. Advantageously, in contrary to the assumption of traditional gossip protocols discussed above, not every node is interested in all the data being disseminated by all the components. Thus, certain application-specific bounds should be established for knowledge dissemination.

For this purpose, we define for each ensemble its *communication group* as the set of nodes to which the ensemble's knowledge dissemination is limited. This set consists of all the nodes where components forming the ensemble are hosted and all the relays necessary for knowledge propagation. Relying on the fact that data is disseminated via gradual flooding, we define a *communication boundary* as the predicate determining the limits of a particular communication group w.r.t. network topology. The relays not satisfying the communication boundary will not participate in the dissemination. In a way, a communication group forms a dynamic, architecture-specific network overlay for knowledge dissemination.

Naturally, a communication boundary includes all the nodes "potentially interested" in the disseminated replicas, while excluding as many of the other nodes as possible. Thus, a communication boundary forms a conservative approximation of the ensemble membership. For example, given the pervasive application from Fig. 1, the communication boundary for the ensemble definition in Fig. 2 can be formulated as follows:

"*For every mission, include all components within all the areas in which the participants of the mission operate.*"

In this example, the communication boundary reflects the fact that all components satisfying the membership condition of the ensemble, i.e., those participating on the same mission, operate in one of the predefined areas. Note however, that the communication boundary predicate is generic w.r.t. a particular mission – it determines a number of different communication groups (thus approximating a number of different ensemble instances), namely a distinct group per distinct mission.

To achieve its desired functionality, a relay has to evaluate a communication boundary much more efficiently than membership condition, preferably using exclusively locally-available information. Thus, we specify communication boundary as a predicate over the local knowledge of the relay and the particular knowledge being disseminated.

Since "communication group" is an application-specific concept relating to application architecture (namely to ensemble membership), we capture it by extending the ensemble definition with a definition of the communication boundary. In addition, we extend the existing concept of "role" to be applicable also at the level of nodes – we say that a node supports a role if one of the components (representative) deployed on the node has structurally-matching knowledge (structural matching enables designing open-ended architectures).

Technically, a communication boundary is defined by a set of predicates (lines 13-16 in Fig. 3). Each of these predicates, given a relay role and a replica role, determines whether a node that has a representative matching the relay role meets the communication boundary for a replica that matches the replica role. Formally, the communication boundary is a conjunction of these predicates (having the form of implications). A relay role has to be either the coordinator or member role.

As an example, in Fig. 3 we show a revised version of the ensemble definition from Fig. 2. Specifically, given a replica corresponding to the member role (TemperatureSensor), the communication boundary includes all relay nodes featuring the TemperatureRelay role, which are in one of the mission areas specified by the replica. This is captured on lines 13-14, which semantically form an implication: the line 13 forms the antecedent (i.e., "if the relay has the role TemperatureRelay and the replica corresponds to the member's role"), while line 14 forms the conclusion. Note, that we have extended the TemperatureSensor role and the knowledge of all the related components to provide the information about mission areas. Similarly, on lines 15-16 the predicate prevents any relaying of replicas matching the coordinator role (as there is no knowledge exchange towards the member). This can be illustrated on Fig. 1 as follows. Provided that all nodes feature the TemperatureRelay role and given that the node 6 participates in a mission that is different to the mission of 9 and localized only to the building #1, then this communication boundary prevents 9 disseminating knowledge of 6 to building #2, as well as 3 from disseminating knowledge of 4. On the other hand, 9, as well as any node in building #1, will disseminate the knowledge of 6 within the building #1. Moreover, 9 will disseminate knowledge of #4 and #7 also to the building #2 via IP.

This part of specification of communication boundary aligns well with the knowledge dissemination in MANETs, where the set of potential recipients is limited by their geographical locality. On the other hand, in large networks that enable routing

```
1.   role TemperatureRelay:
2.      position
3.
4.   role TemperatureSensor:
5.      missionID, missionAreas, temperature
6.
7.   ensemble TemperatureUpdate:
8.      coordinator: TemperatureAggregator
9.      member: TemperatureSensor
10.     membership:
11.        member.missionID == coordinator.missionID
12.     boundary:
13.        case relay: TemperatureRelay, replica: roleOf(member):
14.              ∃area ∈ replica.missionAreas: isInArea(relay.position, area)
15.        case relay: any, replica: roleOf(coordinator):
16.              false
17.        ip-registry: 10.10.16.35, 10.10.16.112
```

Fig. 3. Example of a communication boundary definition in DEECo.

based on global addressing, such as IP networks, a necessary performance optimization is to disseminate replicas only to recipients which themselves meet the communication boundary (rather than blindly pollute the entire IP network). To do this, given a replica, a sender has to be able to (at least partially) assess the validity of the communication boundary with respect to the recipient.

To address this issue, we assume that well-known registries exist providing a relay node the information which other IP-based nodes are part of a communication group (given a particular replica). To avoid unnecessary centralization, such a registry is ensemble specific. The registry either provides statically-defined recipients (well-known relay nodes) or evaluates the communication boundary with respect to a recipient. In the latter case, the potential recipient relay nodes provide the registry with the required relay knowledge. Syntactically, the communication boundary definition contains a set of IP addresses identifying the registries that are specific to the corresponding ensemble (line 17 in Fig. 3). Note that due to the nature of gossip, we do not require all the registries in a given ensemble specification to contain the same information.

3.4 Gossip-based Semantics

To allow for formal analysis of functional and timing properties and precise simulations, as for instance given in Section 4, we have formalized the computational model described in the previous section in terms of operational semantics, which also acts as a thorough, detailed description of the computational model. Technically, based on our previous work [10] we represent the semantics via a state transition system generated by a set of inference rules. Additionally, considering (soft) real-time properties of CPS, the formalization allows only transition traces that are admissible with respect to real-time periodic scheduling of the system processes, ensemble knowledge exchange, and (gossip-based) knowledge dissemination. In a way, these restrictions impose a fairness constraint on the transition traces. Due to space constraints, we refer the interested reader to the technical report [11] for a description of the semantics.

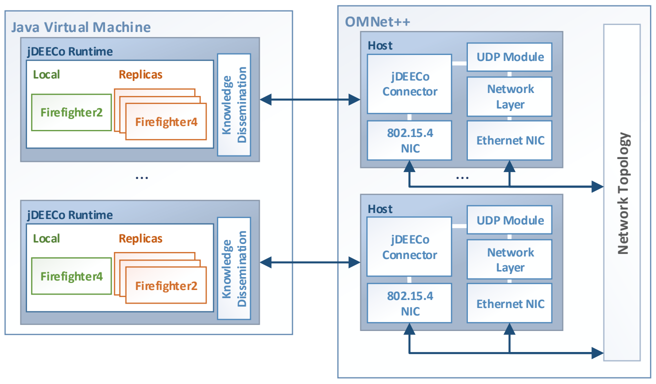


Fig. 4. jDEECo Runtime Framework – OMNet++ integration overview.

4 Implementation

We have implemented[2] the proposed approach by extending the current implementation of jDEECo (a Java implementation of DEECo Runtime). Specifically, we have added support for the concept of communication boundary and the gossip-based knowledge dissemination and ensemble evaluation presented in Section 3. Since these concepts are closely connected to the network layer, we have also integrated jDEECo with the OMNet++ simulation framework[3] that provides an appropriate abstraction for the network infrastructure, enabling precise discrete-time simulations (Fig. 4).

From the perspective of the OSI (Open Systems Interconnection) model [12], our implementation glues together the application layer given by jDEECo Runtime (along with the deployed components and ensembles) with the underlying layers implemented in OMNet++ (Fig. 4). An instance of jDEECo Runtime reflects a single unit of network deployment (e.g., a mobile device). Apart from managing components, scheduling of component processes' execution and ensemble evaluations, jDEECo Runtime automates knowledge management, including network communication needed for knowledge replica dissemination. Each jDEECo Runtime continuously advertises the knowledge of the locally deployed components and, additionally, acts as a relay.

At the network layer, each jDEECo Runtime is bound to its OMNet++ counterpart (namely OMNet host), with which it communicates via JNI (Java Native Interface) calls. Every OMNet host is equipped with two kinds of Network Interface Cards (NICs): one for MANET-based wireless (IEEE 802.15.4) and one for IP-based (Ethernet) communication. Direct communication is implemented via UDP on top of the Ethernet NIC, while MANET-oriented broadcast communication is performed via the wireless NIC. For implementation, we relied on two extensions of OMNet++:

[2] https://github.com/d3scomp/JDEECo
[3] http://omnetpp.org/

the MiXiM plugin delivering a detailed model of the 802.15.4 protocol and the INET framework implementing the whole Ethernet stack.

Each jDEECo Runtime gossips knowledge replicas obtained from the network. We specifically distinguish two cases: gossiping via MANET and direct gossiping. In the case of MANET gossiping, a jDEECo Runtime calculates a probabilistic rebroadcast delay relying on RSSI (Radio Signal Strength Indicator); in case of direct gossiping the data is retransmitted to a random set of peers using a fixed delay. To prevent network overload, the rebroadcast is aborted in case a newer replica is received from another peer. Additionally, MANET gossiping is aborted if the same replica comes from the MANET NIC. The delay and aborting mechanism of MANET gossip is based on the counter-based algorithm proposed in [6].

5 Evaluation

In this section, we show that our gossip-based ensemble evaluation is practically feasible by providing measurements that answer the following fundamental questions: (1) how the gossip-based ensemble evaluation scales with respect to the number of nodes in the system, and (2) how the communication boundary improves the scalability. Specifically, we do it by simulation and measurements of the motivating scenario model.

Building on the implementation outlined in Sections 2 and 3, the evaluated scenario consists of several deployed firefighter teams that partially overlap in terms of radio signal coverage. Each team uses the other teams' members as relays for knowledge dissemination in the overlapping areas to ensure the necessary wireless coverage. The objective of this scenario is to illustrate the performance gain of employing communication boundary, which limits data sharing strictly to the overlapping regions. Note that the communication boundary being used (Fig. 3) allows any node that monitors its position, such as a device of other emergency personnel, to be equally included into the scenario and act as a relay; for brevity we include only firefighters. The scenario combines MANET-based gossiping (with evenly distributed nodes in the area) and direct gossiping realized by Ethernet-enabled nodes (a small fraction of the nodes).

The scenario is affected a large number of factors, such as network density, size of the overlapping regions, wireless communication range, gossip protocol configuration, etc. Therefore, we have simulated our system under a variety of configurations; however, due the space limits, this paper presents results for configurations varying in the number of overlapping teams (thus also in the total number of nodes), while maintaining a fixed node density (close to the highest density safely manageable by the implemented MANET gossip protocol, as evaluated by Williams and Camp in [4]). The detailed information on the configuration parameters, which were set to match the realistic case described in Section 2 as close as possible, as well as the simulation results for various set-ups, can be found on the DEECo project website[4].

The results presented in Fig. 5 show the leader-member end-to-end communication time in a firefighting team (in particular, the time it takes a leader node to learn that a

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

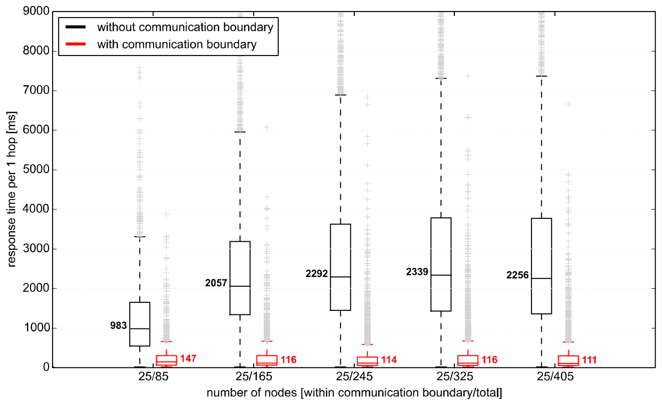


Fig. 5. Time for discovering a team Member in danger by a corresponding Leader.

member of its team is in danger, normalized by the hop distance between the two nodes). Specifically, we compare the cases with and without communication boundary. Not using communication boundary results into propagation of a team's data across all nodes; this causes global degradation of end-to-end communication performance (corresponding to the performance limitations of the implemented gossip protocol). On the other hand, communication boundary localizes the team's data dissemination and prevents the communication channels from overloading, which results in stable performance (as long as the dynamic communication boundary does not grow). Specifically, the communication boundary reduces the utilization of the shared communication medium by preventing ”outside” data from penetrating deeper (than necessary) into the team's area. This reduces the overhead of the communication medium; the freed capacity can be now utilized to handle dissemination of the team's data.

6 Discussion

In this section we review the key contributions of our approach and discuss the main related challenges that stem from the decentralized decisions on ensemble membership and gossip-based communication.

6.1 Key Contributions

Integrating the DEECo concept of ensemble with gossip-based communication enables for efficiently dealing with scenarios where system architecture is open-ended and changes continuously; e.g., systems with high mobility of components or largely unreliable communication links. To this end, the autonomicity of DEECo components and best-effort style of communication provided by the gossip-based implementation of ensemble knowledge exchange deliver means for assuring high infrastructural resilience.

Although, due to the dynamic nature of ensembles, the gossip-based implementation of knowledge exchange requires disseminating knowledge to all the potential members, possibly requiring all nodes to act as relays, communication boundary provides means to accurately reduce the dissemination to only those nodes, which are actually needed considering the application-logic point of view. Moreover, as the knowledge dissemination governed by the communication boundary exploits the contextual information available at the application level in the form of component knowledge (current position, temperature etc.), the possible set of relay nodes may change dynamically according to data being disseminated and the state of the relay nodes, as opposed to generic indicators for limiting communication, such as timestamps and hop count.

Consequently, by accurately preventing data from flowing to irrelevant parts of the system, the proposed communication boundary mechanism considerably improves the utilization of the shared communication medium within the MANET network. The gain in communication performance depends on how accurate estimate of a membership the relevant communication boundary is.

6.2 Related Challenges

Belief inaccuracy in asynchronous knowledge dissemination. The belief a component has about the knowledge of another component is essentially always outdated. This outdatedness is mainly rooted in (i) network infrastructure performance (e.g., bandwidth, packet delays, medium access rate, etc.) (ii) MANET topology issues (e.g., large hop distance between sender and receiver), and (iii) ineffective tuning of the employed gossip algorithm (e.g., too long (re)transmission period).

The outdateness of belief determines its inaccuracy, i.e. the difference between the value of the belief and the actual value of the knowledge. Depending on the nature of data (i.e., continuous or discrete domain, rate of change), slight incoherence between knowledge and belief might be tolerated or accounted for during design [8]. Advantageously, this is the case with most of CPS where real-world phenomena (e.g., position, oxygen level, velocity) are to be captured.

Split-brain situations in ensembles. Due to the belief outdatedness and isolated membership evaluation by each potential member, situations where different nodes arrive at conflicting conclusions regarding ensembles may arise. This results in a member acting as if it were in an ensemble having a coordinator who is not aware of it (or vice-versa). As an example, consider an ensemble that is formed of the firefighter components (each hosted on a separate node) whose positions lie within a 10-meter perimeter from a leader (coordinator). When a firefighter node steps out of the designated area, the corresponding firefighter component should not be part of the ensemble. The coordinator, however, will only learn about that at the next time its host node receives an up-to-date replica of that component. Until then, it will falsely consider the firefighter component as a legitimate member of its ensemble.

In cases where belief outdatedness and topology dynamicity are not too high these “split-brain” situations are of temporal nature. For deeper analyses, system simulations (see Section 4) and timing analysis can be used to provide measurements of the distribution of such inconsistencies and their duration.

Gossip implementation. For our experiments we employed a basic version of counter-based gossiping [6] without emphasis on its optimization, as we did not intend to evaluate the gossip protocol per se but rather the practical feasibility of gossiping ensembles and the impact of the communication boundary. One of such optimizations of the communication that we identified as an absolute necessity was stripping down the size of the disseminated replicas. This is especially critical in MANET settings, where the bandwidth is limited and larger replicas (more than approx. 128 bytes) lead to fragmentation. In combination with the CSMA/CA medium access technique and the hidden node problem [13] this leads quickly to network contention.

7 Related work

The solution presented in this paper brings about the convergence of software component models for CPS and gossip-based communication. Although there are some attempts to achieve synergy between the two areas ([14–16]), they are set on a significantly different track than our approach. In [14], the authors propose a conceptual architecture and design framework for gossip. The framework is based on reusable building blocks, where individual protocols are treated as monolithic black boxes. In [15], the authors propose an API for programming gossip-based systems by analyzing the identified recurrent design dimensions of gossip protocols – namely randomness, neighborhood, and communication. Finally, in [16], the authors introduce a component framework GossipKit, which aims at facilitating the development and testing of gossip protocols by relying on reusable and modular gossip abstractions and standard component-based composition techniques. In all of these approaches the focus is on providing an architectural solution for building gossip-based middleware by means of ready-made components/interfaces. We, in contrast, focus on modeling application logic by means of autonomous components which use gossip internally and partially transparently as the primary means of their communication.

Regarding the state of the art in gossip-based communication, different variations of the basic gossiping scheme have been proposed for different application domains and with slightly different semantics ([17–19]). In MANETs gossiping translates to probabilistic broadcasting within the wireless range of each node [3]. Probabilistic forwarding is often combined with some other locally computable mechanism, such as counter-based [6], location-based [20], distance-based [21], energy-based [22], or a combination of these, to further reduce the number of retransmitted messages (with respect to blind flooding). In our work we do not intend to extend or evaluate the state of the art in gossip-based communication, but provide a method for architecting CPS using abstractions that facilitate the efficiency of the gossip by relying on the architecture-level context information.

Regarding component models and architectures supporting distributed dynamic systems such as CPS, different approaches related to self-adapting/self-organizing systems [23, 24], self-managing architectures [25], component-based architectures [26, 27], and architectural models at runtime [28] have been proposed. The common denominator of these approaches is the fact that they do not support high dynamicity (which does not scale with the ever-changing landscape of CPS) or they do not readily map to decentralized architectures. DEECo, on the other hand, fits better the specifics of CPS by relying on dynamic component grouping and implicit component communication.

8 Conclusions

In this paper, we presented a synergy of software component model abstractions and gossip-based communication primitives as a promising solution for engineering scalable dynamic decentralized cyber-physical systems. Our approach relies on providing architecture-level descriptions that feature communication groups (captured by communication boundaries) and allow us "driving" the gossip efficiently. The presented experiments show that our approach is in principle feasible. Our current and future work involves improving the scalability of our approach by various optimizations of the gossip protocol (e.g., employing location-based algorithms where GPS-enabled devices are required). Another direction is investigating timing constraints on the gossip-based knowledge dissemination and exchange which will supplement the strict real-time constraints already imposed on local component behaviors.

Acknowledgments. 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. Beetz, K., Böhm, W.: Challenges in Engineering for Software-Intensive Embedded Systems. Model-Based Engineering of Embedded Systems. pp. 3–14. Springer (2012).
2. Lee, E.A.: Cyber Physical Systems: Design Challenges. Proc. of ISORC'08. pp. 363–369. Orlando, FL, USA (2008).
3. Friedman, R., Gavidia, D., Rodrigues, L., Viana, A.C., Voulgaris, S.: Gossiping on MANETs: the Beauty and the Beast. ACM SIGOPS Oper. Syst. Rev. 41, 67–74 (2007).
4. Williams, B., Camp, T.: Comparison of Broadcasting Techniques for Mobile Ad Hoc Networks. Proc. of MobiHoc'02. pp. 194–205. ACM, Lausanne, Switzerland (2002).
5. Eugster, P.T., Guerraoui, R., Handurukande, S.B., Kouznetsov, P., Kermarrec, A.-M.: Lightweight probabilistic broadcast. ACM TOCS. 21, 341–374 (2003).
6. Tseng, Y.-C., Ni, S.-Y., Chen, Y.-S., Sheu, J.-P.: The Broadcast Storm Problem in a Mobile Ad Hoc Network. Wirel. Networks. 8, 153–167 (2002).
7. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: DEECo – an Ensemble-Based Component System. Proc. of CBSE'13. pp. 81–90. ACM, Vancouver, Canada (2013).
8. Ali, R. Al, Bures, T., Gerostathopoulos, I., Keznikl, J., Plasil, F.: Architecture Adaptation Based on Belief Inaccuracy Estimation. To appear in Proc. of WICSA'14 (2014).

9. Drabkin, V., Friedman, R., Kliot, G., Segal, M.: RAPID: Reliable Probabilistic Dissemination in Wireless Ad-Hoc Networks. In Proc. of SRDS'07. pp. 13–22. IEEE, Beijing, China (2007).
10. Barnat, J., Benes, N., Bures, T., Cerna, I., Keznikl, J., Plasil, F.: Towards Verification of Ensemble-Based Component Systems. To appear in Proc. of FACS'13. Springer, Nanchang, China (2013).
11. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: Computational Model for Gossiping Components in Cyber-Physical Systems. Charles University in Prague, TR no. D3S-TR-2014-03.
12. OSI: OSI Basic Reference Model: The Basic Model - ISO/IEC 7498-1, http://standards.iso.org.
13. Yoo, J., Kim, C.: On the Hidden Terminal Problem in Multi-rate Ad Hoc Wireless Networks. Information Networking, v. 3391 of LNCS. pp. 479–488. Springer (2005).
14. Rivière, E., Baldoni, R., Li, H., Pereira, J.: Compositional gossip: a conceptual architecture for designing gossip-based applications. ACM SIGOPS Oper. Syst. Rev. 41, 43–50 (2007).
15. Eugster, P., Felber, P., Le Fessant, F.: The "Art" of Programming Gossip-based Systems. ACM SIGOPS Oper. Syst. Rev. 41, 37–42 (2007).
16. Taiani, F., Lin, S., Blair, S.G.: GossipKit: A Unified Component Framework for Gossip. IEEE Trans. Softw. Eng. PP, 1–17 (2013).
17. Branco, M., Leitão, J., Rodrigues, L.: Bounded Gossip: A Gossip Protocol for Large-Scale Datacenters. Proc. of SAC'13. pp. 591–596. ACM, Coimbra, Portugal (2013).
18. Khelil, A., Suri, N.: Gossiping: Adaptive and Reliable Broadcasting in MANETs. Dependable Computing, v. 4746 of LNCS. pp. 123–141. Springer (2007).
19. Kermarrec, A.-M., Van Steen, M.: Gossiping in distributed systems. ACM SIGOPS Oper. Syst. Rev. 41, 2–7 (2007).
20. Karp, B., Kung, H.T.: GPSR: Greedy Perimeter Stateless Routing for Wireless Networks. Proc. of MobiCom'00. pp. 243–254. ACM, Boston, USA (2000).
21. Cartigny, J., Simplot, D.: Border Node Retransmission Based Probabilistic Broadcast Protocols in Ad-Hoc Networks. Proc. of HICSS'03. pp. 303–312. IEEE, Hawaii, USA (2003).
22. Miranda, H., Leggio, S., Rodrigues, L., Raatikainen, K.: A Power-Aware Broadcasting Algorithm. Proc. of PIMRC'06. pp. 1–5. IEEE, Helsinki, Finland (2006).
23. Serugendo, G.D.M., Fitzgerald, J., Romanovsky, A.: MetaSelf – An Architecture and a Development Method for Dependable Self- * Systems. Proc. of SAC'10. pp. 457–461. ACM, Sierre, Switzerland (2010).
24. Liu, H., Parashar, M., Hariri, S.: A Component Based Programming Framework for Autonomic Applications. Proc. of ICAC'04. pp. 10–17 (2004).
25. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. Proc. of FOSE'07. pp. 259–268. IEEE, Minneapolis, USA (2007).
26. Baresi, L., Guinea, S., Tamburrelli, G.: Towards Decentralized Self-adaptive Component-based Systems. Proc. of SEAMS'08. pp. 57–64. ACM, Leipzig, Germany (2008).
27. Peper, C., Schneider, D.: Component engineering for adaptive ad-hoc systems. Proceedings of SEAMS '08. pp. 49–56. ACM, Leipzig, Germany (2008).
28. Morin, B., Barais, O., Jezequel, J.-M., Fleurey, F., Solberg, A.: Models at Runtime to Support Dynamic Adaptation. Computer (Long. Beach. Calif). 42, 44–51 (2009).

4.4 Computational Model for Gossiping Components in Cyber-Physical Systems

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

Technical Report D3S-TR-2014-03.

Report No: D3S-TR-2014-03,
April 2014.

The original version is available electronically from the department site at http://d3s.mff.cuni.cz/publications/download/D3S-TR-2014-03.pdf.

Summary of the Paper

This is a technical report specifying the computational model showing an intersection between high-level architectural models and low-level communication models to enable application-specific communication in component-based systems. The report targets MANET deployment of a DEECo-based system and aims to provide an understanding and argumentation on the approaches selected for implementation. Section 2 describes the core idea of the gossip-based communication and ensemble realization at the level of the jDEECo platform. In addition, it includes the description of the communication boundary introduced in the paper from Section 4.3. Following, the formal specification of the component behavior and ensemble evaluation is given. There, the details of component internals (i.e. knowledge and processes) as well as ensemble constituents (i.e. membership condition and knowledge exchange) are expressed in a formal way. Using this representation, the asynchronous communication process in DEECo is described enabling potential formal analytical methods to be applied. Specifically, the methods that would verify the system against its real-time properties given in the system requirements.

Author Contribution and Goals Addressed

Author contribution in this work includes participation in the main idea of the communication process formulation as well as its validation by delivering its realization in the form of the jDEECo platform.

Considering the research goals addressed in this work, the goal **G2** is addressed by this formalism. It complements the implementation of the jDEECo platform with its formal description allowing for early stage system analysis and ensuring both correct deployment and execution of the designed system.

Department of Distributed and Dependable Systems
Technical Report no. D3S-TR-2014-03
April 2nd, 2014

Computational Model for Gossiping Components in Cyber-Physical Systems

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

{bures, iliasg, hnetynka, keznikl, kit, plasil}@d3s.mff.cuni.cz

Abstract: Developing software for dynamic cyber-physical systems (CPS) is a complex task A particular challenge in this context is the robust distributed data dissemination in dynamic networks. Gossip-based communication stands as a promising solution to this challenge. We argue, that exploitation of application-specific information, software architecture in particular, has a large potential for improving the robustness and performance of gossip-based communication. This report presents a computational model that represents a synergy between high-level architectural models and low-level communication models to effectively enable application-specific gossiping in component-based systems.

Version: April 2nd, 2014

This work was partially supported by the EU project ASCENS 257414. 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.

1 Introduction

Cyber-physical systems (CPS) are complex networked systems where the interplay of software control with the physical environment has a prominent role. Examples range from intelligent navigation systems (cars can communicate with each other and with street infrastructure units to minimize traffic congestion, fuel consumption, etc.) to emergency coordination systems and interactive distributed games. Modern CPS are inherently distributed and large-scale, and consist of both stationary and mobile devices. They are also increasingly depending on software which has actually become their most intricate and extensive constituent [3].

Whereas the challenges and opportunities of CPS cover a range of areas, in this report we focus on the communication requirements of CPS. The main observation is that communication in CPS has to be robust in spite of frequent connection faults, transient network failures, and inherently unreliable communication mediums (e.g., wireless). At the same time, communication primitives should reflect and ideally take advantage of the CPS specifics, such as physical mobility and partitioning.

Looking at the state-of-the-art in distributed systems communication, gossip or epidemic protocols provide an efficient way to address the aforementioned specifics. Gossip protocols cope with node and network failures, are scalable due to their symmetric nature, and can exploit the physical mobility of gossiping nodes [5]. Gossiping paradigm has already been applied with success in both Internet-based systems and wireless mobile ad-hoc networks (MANETs) [7]. The gossip protocols typically combine probabilistic forwarding with counter-based, distance-based, and location-based mechanisms. These mechanisms and configuration parameters are, however, only available at the lower level of the software stack, often transparent to the application/architecture layer, which is problematic when the spread of data depends on the architectural configuration in question.

In this report, we aim at bridging the abstraction gap between gossip protocols (concerned with message sending/receiving, packet-based communication, peer selection, etc.) and application-level programming using component models and architectures tailored to the specifics of CPS. Specifically, we do it by encompassing gossiping primitives into sound software engineering abstractions, which allow for (i) systematic engineering of CPS via gossiping components and (ii) application-specific, scalable, and efficient gossip-based communication. We do so in the context of DEECo [4] – a component model that specifically supports dynamic, ever-changing architectures and reflects the specifics of CPS within its abstractions by relying on the concepts of autonomous real-time components, and dynamic ad-hoc coordination groups (ensembles). For further information on DEECo, we refer an interested reader to [4, 6].

2 Gossiping in ensembles

To integrate the communication mechanisms of gossip-based protocols into DEECo, we have adopted a fully decentralized and robust approach relying on gossip for establishing ensembles and performing knowledge exchange. In principle, we replace the network communication layer of DEECo by gossip-based communication and extend the DEECo architectural model (the definition of ensembles in particular) by the concept of a *communication boundary* so as to allow efficient functioning of the underlying gossip. In this section, we describe our approach informally and formalize it in Section 3.

To connect components at the architectural level with their physical deployment, we define node as a hardware/software platform where a number of DEECo components are deployed. Each node contains an instance of Runtime, taking the role of component container and communication middleware. A node contains both the knowledge of hosted components and copies (replicas) of the knowledge of components hosted on other nodes. Nodes communicate with each other via their network interfaces depending on the available networking infrastructure. Thus, component communication is constrained by the available networking infrastructure between the nodes the components are deployed on. Inspired by the motivating scenario, we focus on combinations of IP-based networks (wireless and wired) and MANET networks (which allow only for short range direct communication).

The main principles of our approach can be characterized by the following rules pertaining to every node:

(1) *A node has its own awareness of ensemble instances existing in the system, specifically of those that contain components deployed on the node. This is based on evaluating the membership predicates with respect to the knowledge of (a) local components and (b) replicas.*
(2) *A node performs knowledge exchange independently, based on the knowledge of both local components and replicas.*
(3) *A node proactively disseminates component knowledge, so that every other node has the knowledge relevant for realization of (1) and (2).*

In the following, the individual elements of our approach are described in more detail – points 1 and 2 are explained in Section 2.1and point 3 is elaborated in Sections 2.2 and 2.3.

2.1 Decentralized evaluation of ensemble membership and knowledge exchange

Instead of forming ensembles by looking at a snapshot of the whole system (which would imply that a global view on the system has to be available), we take a node-centric approach. Every node periodically iterates over all known ensemble definitions and checks whether a local component can act as a member or coordinator in an instance of the ensemble definition, given its replicas. For each such ensemble instance, it performs the corresponding knowledge exchange, which results in updating the local components' knowledge (but not the replicas).

As an example, consider an instance of the `TemperatureUpdate` ensemble definition show in Fig. 1, evaluated on the site of the coordinator. In this case, the knowledge exchange results into updating the coordinator's field temperatures.

Note that a consequence of this technique is that degradation of system performance when no connectivity is available is gradual: each Runtime effectively operates on the locally available replicas until they become too outdated to rely upon. Here, we count on one of the specifics of CPS, namely that values of most magnitudes (e.g., temperature in Fig. 1) evolve gradually according to physical laws [1]. Practically this means that a belief which is not too old may still be at least partly relevant.

Another consequence is that, due to belief outdatedness causing belief inaccuracy, it is possible for a component to behave as if it were in ensemble with a coordinator, which is not aware of it (and vice-versa).

2.2 Asynchronous knowledge dissemination via gossip

The decentralized solution presented in Section 2.1, requires that each node possesses all the necessary replicas from the components that can potentially participate in ensembles with its local components. We enable this by asynchronous gossip-based knowledge dissemination between all the components of a DEECo application.

The main idea is that every node periodically publishes the knowledge of its local components on the network. For MANETs, this translates to periodic broadcast within the wireless range of the node. For IP networks, it translates to periodic sending to randomly selected nodes. Upon reception of a component's knowledge, every node probabilistically decides whether to retransmit the received knowledge. The nodes that perform such re-transmission are then acting

```
role TemperatureRelay:
  position

role TemperatureSensor:
  missionID, missionAreas, temperature

ensemble TemperatureUpdate:
  coordinator: TemperatureAggregator
  member: TemperatureSensor
  membership:
    member.missionID == coordinator.missionID
  knowledge exchange:
    coordinator.temperatures ← { (m.ID, m.temperature) | m ∈ members }
  boundary:
    case relay: TemperatureRelay, replica: roleOf(member):
        ∃area ∈ replica.missionAreas: isInArea(relay.position, area)
    case relay: any, replica: roleOf(coordinator):
        false
    ip-registry: 10.10.16.35, 10.10.16.112
  …
```

Fig. 1. Example of an ensemble definition in DEECo, extended with a communication boundary.

as relays. Here, we rely on the probabilistic convergence of gossip protocols which ensures that every node will eventually receive the knowledge of every component in a bounded number of steps.

Note that this dissemination scheme dictates that retransmission is potentially done by all nodes, not only from the ones that are interested in the disseminated knowledge (i.e., nodes hosting components that could be members of the ensemble which the disseminated knowledge relates to).

2.3 Bounding the gossip

Although the aforementioned gossip-based knowledge dissemination successfully conveys the knowledge of all nodes, it raises performance issues. Specifically, a DEECo application is considered as a ubiquitous ecosystem in a real environment, the application is potentially boundless w.r.t. network reachability. In such a system, unlimited gossiping is not a viable option. Advantageously, in contrary to the assumption of traditional gossip protocols discussed above, not every node is interested in all the data being disseminated by all the components. Thus, certain application-specific bounds should be established for knowledge dissemination.

For this purpose, we define for each ensemble its communication group as the set of nodes to which the ensemble's knowledge dissemination is limited. This set consists of all the nodes where components forming the ensemble are hosted and all the relays necessary for knowledge propagation. Relying on the fact that data is disseminated via gradual flooding, we define a communication boundary as the predicate determining the limits of a particular communication group w.r.t. network topology. The relays not satisfying the communication boundary will not participate in the dissemination. In a way, a communication group forms a dynamic, architecture-specific network overlay for knowledge dissemination.

Naturally, a communication boundary includes all the nodes "potentially interested" in the disseminated replicas, while excluding as many of the other replicas as possible. Thus, a communication boundary forms a conservative approximation of the potential ensemble membership. For example, the communication boundary for the ensemble definition in Fig. 1 can be formulated as follows:

"For every mission, include all components within all the areas
in which the participants of the mission operate."

In this example, the communication boundary reflects the fact that all components satisfying the membership condition of the ensemble, i.e., those participating on the same mission, operate in one of the predefined areas. Note however, that the communication boundary predicate is generic w.r.t. a particular mission – it determines a number of different communication groups (thus approximating a number of different ensemble instances), namely a distinct group per distinct mission.

To achieve its desired functionality, a relay has to evaluate a communication boundary much more efficiently than membership condition, preferably using only locally-available information. Thus, we specify communication boundary as a predicate over the local knowledge of the relay and the particular knowledge being disseminated.

Since "communication group" is an application-specific concept relating to application architecture (namely to knowledge of a component), we extend the ensemble definition to capture communication group by definition of communication boundary. In addition, we extend the existing concept of role to apply at the level of nodes so that a node supports a role if one of the components (representative) deployed on the node has structurally-matching knowledge. The structural matching enables designing open-ended architectures.

Technically, a communication boundary is defined by a set of predicates (lines 14-17 in Fig. 1). Each of these predicates, given a relay role and a replica role, determines whether a node that has a representative matching the relay role meets the communication boundary for a replica that matches the replica role. Formally, the communication boundary is a conjunction of these predicates (having the form of implications). A relay role has to be either the coordinator or member role.

For example in Fig. 1, given a replica corresponding to the member role (TemperatureSensor), the communication boundary includes all relay nodes featuring the TemperatureRelay role, which are in one of the mission areas specified by the replica. This is captured on lines 15-16, which semantically form an implication: the line 15 forms the antecedent (i.e., "if the relay has the role TemperatureRelay and the replica corresponds to the member's role"), while line 16 forms the conclusion. Note, that we have extended the TemperatureSensor role and the knowledge of all the related components to provide the information about mission areas. Similarly, on lines 17-18 the predicate prevents any relaying of replicas matching the coordinator role (as there is no knowledge exchange towards the member).

This part of specification of communication boundary aligns well with the knowledge dissemination in MANETs, where the set of potential recipients is limited by their geographical locality. On the other hand, in large networks that enable routing based on global addressing, such as IP networks, a necessary performance optimization is to disseminate replicas only to recipients which themselves meet the communication boundary (rather than blindly pollute the entire IP network). To do this, given a replica, a sender has to be able to (at least partially) assess the validity of the communication boundary with respect to the recipient.

To address this issue, we assume that well-known registries exist providing a relay node the information which other IP-based nodes are part of a communication group (given a particular replica). To avoid unnecessary centralization, such a registry is ensemble specific. The registry either provides statically-defined recipients (well-known relay nodes) or evaluates the communication boundary with respect to a recipient. In the latter case, the potential recipient relay nodes provide the registry with the required relay knowledge. Syntactically, the communication boundary definition contains a set of IP addresses identifying the registries that are specific to the corresponding ensemble (line 19 in Fig. 1). Note that due to the nature of gossip, we do not require all the registries in a given ensemble specification to contain the same information.

3 Formalized Semantics

Having informally outlined our approach in the previous section, we provide now a precise formulation of our solution. We resort to formalization using operational semantics, which allows us to do analysis of functional and timing properties and precise simulations.

Technically, we represent the semantics via a state transition system, where the transitions are generated by the inference rules presented in the rest of this section. Specifically, each rule defines the required state in which the transition is allowed and the state change entailed by the transition.

3.1 Initial definitions

Knowledge is a partial function $knw: F \nrightarrow D$, mapping the domain of knowledge fields F (typically seen as identifiers) to the set of knowledge field values D. We denote K as the set of all possible knowledge valuations.

Knowledge update is a partial function $u: F \nrightarrow (D \cup \{undef\})$, where undef represents a special value which signifies that a knowledge field should be removed from a knowledge. We denote U as the set of all knowledge updates. We further define the knowledge update operator $\oplus$ with the following semantics. Let $knw \in K$, $u \in U$, and $knw' = knw \oplus u$, then value of $knw'(f)$ (for a knowledge field f) is:

- $knw(f)$ if $u(f)$ is not defined; or
- $u(f)$ if $u(f) \neq$ undef ; or
- $\perp$ (not defined) if $u(f) = undef$.

3.2 Internal component behavior

Component is a tuple $c = (knw, proc, rproc)$, where knw is the knowledge of the component, $proc$ is the set of processes of the component, and $rproc$ is the state of running processes (definitions follow below). Note, that a component is understood as a singleton, i.e., a component instance. We denote the set of all components as C.

- **Process** of a component c is a function $p: K \times R \rightarrow U$, where R denotes the domain of internal events which are not explicitly modeled on the level of the component model and thus from the perspective of the semantics remain non deterministic (e.g., responses from the operating system, readings from sensors, etc.).
- The **state of running processes** is a partial function $rproc: proc \nrightarrow K$, which maps each currently running process to its inputs (i.e., valuation of the knowledge at the time of the process start). If a process is not running, the value of $rproc$ is undefined.

The inference rules below describe the semantics of the concurrent, asynchronous component process execution:

$$\frac{c \in C,\ p \in c.proc,\ c.rproc(p) = \perp}{c.rproc(p) \leftarrow c.knw} \qquad \text{(p-start)}$$

$$\frac{\begin{array}{c} c \in C,\ p \in c.proc \\ c.rproc(p) \neq \perp,\ k = c.rproc(p),\ r \in R \end{array}}{c.rproc(p) \leftarrow \perp,\ c.knw \leftarrow c.knw \oplus p(k,r)} \qquad \text{(p-end)}$$

Each process executes in a cyclic manner. This is modeled in two steps: (1) inputs of the process are atomically retrieved from component's knowledge (rule p-start), and (2) outputs of the process computation are atomically written to component's knowledge (rule p-end). Essentially, this semantics of process execution is similar to our previous work [2].

Note that we use a shorthand form of the inference rules conclusions. We always assume the conclusion to be in form a state transition $S \longrightarrow S'$, where S' is the same state as S, except for assignment explicitly stated in the rule. For instance $c.knw \leftarrow c.knw \oplus u$ means S' is the same as S, except for the knowledge of component c, whose new valuation is $c.knw \oplus u$. We use the assignment operator $\leftarrow$ also to remove an assignment from a partial function – i.e., $f(x) \leftarrow \perp$ means $f \setminus \bigcup_{\forall y}(x,y)$.

3.3 Decentralized execution of ensemble membership and knowledge exchange

Ensemble definition is a tuple $e = (mem, kex, cb)$, where mem is the membership predicate, kex is the knowledge exchange function, and cb is the communication boundary predicate (definitions of membership and knowledge exchange follow below, communication boundary is elaborated in Section 3.4). We denote the set of all ensemble definitions as E.

- **Membership** is a predicate $mem \subseteq K \times K$ over knowledge of an ordered pair of components. The predicate determines whether the two components form the coordinator-member pair of an ensemble instantiated from e.
- **Knowledge exchange** is a function $kex: K \times K \to U \times U$, which for knowledge of the coordinator and a member (in this order) gives the knowledge updates corresponding to the effect of the knowledge exchange in ensemble definition.

Computational node is a tuple $n = (comp, cknw, rkex)$, where $comp \subseteq C$ denotes the set of components deployed on the node, $cknw$ the knowledge replicas available to the node, and $rkex$ is the state of running ensemble knowledge exchange (definitions follow below). We denote the set of all nodes as N.

D3S Technical Report no. D3S-TR-2014-03

- The **knowledge replicas of the node** n is a partial function $cknw: C \nrightarrow K$, which for a component returns the replica of the component's knowledge available to the node n. In case the component c is deployed on the node (i.e, $c \in n.comp$), then $cknw(c) = c.knw$ (i.e., the replica is equal to the actual knowledge of the component). If the node has no available replica for a component, the value is undefined.
- The **state of running ensemble knowledge exchange** is a partial function $rkex : E \times C \times C \nrightarrow K \times K$, which maps each currently running knowledge exchange to its inputs – knowledge of the coordinator and a member at the time of the knowledge exchange start. If knowledge exchange is not running, the value of $rkex$ is undefined.

The inference rules below describe the semantics of the concurrent, asynchronous, and decentralized execution of knowledge exchange in scope of a single node, as described in Section **Error! Reference source not found.**:

$$\frac{n \in N,\ c_l \in n.comp,\ e \in E,\quad c_p \in C{:}\, n.cknw(c_p) \neq \bot,\ n.rkex(e, c_l, c_p) = \bot}{c_l.rkex(e, c_l, c_p) \leftarrow \big(n.cknw(c_l), n.cknw(c_p)\big)} \qquad \text{(e-start)}$$

$$\frac{n \in N,\ c_l \in n.comp,\ e \in E,\quad c_p \in C{:}\, n.rkex(e, c_l, c_p) \neq \bot = (k_l, k_p)}{\begin{array}{c} c_l.knw \leftarrow c_l.knw \oplus \begin{cases} [e.kex(k_l, k_p)]_1, & e.mem(k_l, k_p) \\ [e.kex(k_p, k_l)]_2, & e.mem(k_p, k_l) \\ \emptyset & , \text{otherwise} \end{cases} \\ n.rkex(e, c_l, c_p) \leftarrow \bot \end{array}} \qquad \text{(e-end)}$$

Specifically, the rule (e-start) describes the atomic read of knowledge exchange inputs at the start of its execution. Recall, that knowledge exchange is executed for all pairs of components in which one component (the local component, c_l) is deployed at the node itself while there is an available replica for the other (the peer component, c_p). Note that the peer component can be deployed on the node as well. The rule (e-end) describes the atomic update of the local component's knowledge with the outcome of the process w.r.t. to the stored inputs, performed at the end. Note that the local component's knowledge is actually updated only if the local and peer components meet the membership condition of the corresponding ensemble. If the local component acts in the knowledge exchange as the coordinator, the first knowledge update from the tuple returned by kex is used; if it acts as a member, the second is used.

3.4 Asynchronous knowledge dissemination

We abstract the network as a complete directed graph of nodes, where each edge corresponds to a unidirectional **communication channel** between two nodes. The communication channel is modeled as an unbounded, lossless queue. We denote the set of all channels as Q. A channel may either represent a MANET communication link or an IP communication link. For convenience, we use the function $chan: \{manet, ip\} \times N \times N \to Q$ which for each channel type and each pair of nodes returns a distinct channel connecting the first node with the second. The communication over the channels is regulated by two conditions described below.

The **gossiping condition** is a predicate $gossip \subseteq \{manet, ip\} \times N \times N \times R$, which for a channel type, a sender node, and a recipient node establishes whether the sender should disseminate replicas to the recipient via a channel of that type, based on the employed gossiping scheme and network topology. For example, in case of a MANET channel, this predicate reflects whether the recipient is within the receiving range of the sender, etc. R denotes the domain of internal events which are not explicitly modeled on the level of the component model and thus from the perspective of the semantics remain non deterministic (e.g., random algorithmic decisions, lost packets due to limited range and conflicts in wireless communication). Note, that the gossiping condition is system-specific (depending of the system's network topology etc.).

Ensemble **communication boundary** is an ensemble-specific predicate $cb \subseteq N \times K$, which for a node and a component's knowledge replica establishes whether the node meets the communication boundary w.r.t. the replica; i.e., whether the replica should be disseminated by the node for the purpose of ensemble membership and knowledge exchange evaluation. Recall, that for every ensemble e, $e.mem \subseteq e.cb$. The inference rules below describe asynchronous, gossip-based dissemination of knowledge as described in Section **Error! Reference source not found.** and Section **Error! Reference source not found.**:

$$\frac{\begin{array}{c} r \in R, n_l, n_r \in N\colon\ n_l \neq n_r, c \in C,\ k_C = n_l.cknw(c) \\ (k_1, \dots, k_n) = chan(manet, n_l, n_r), \\ gossip(manet, n_l, n_r, r), \exists e \in E\colon\ e.cb(n_l, k_c) \end{array}}{chan(manet, n_l, n_r) \leftarrow (k_1, \dots, k_n, k_c)} \quad \text{(manet-send)}$$

$$\frac{\begin{array}{c} r \in R, n_l, n_r \in N\colon\ n_l \neq n_r, c \in C,\ k_C = n_l.cknw(c) \\ (k_1, \dots, k_n) =\ chan(ip, n_l, n_r),\ gossip(ip, n_l, n_r, r) \\ \exists e \in E\colon\ e.cb(n_l, k_c) \wedge e.cb(n_r, k_c) \end{array}}{chan(ip, n_l, n_r) \leftarrow (k_1, \dots, k_n, k_c)} \quad \text{(ip-send)}$$

$$\frac{\begin{array}{c} n_l, n_r \in N,\ n_l \neq n_r, c \in C,\ k_c = n_l.cknw(c) \\ t \in \{manet, ip\}, chan(t, n_r, n_l) = (k_c', k_1, \dots, k_n) \neq \perp \end{array}}{\begin{array}{c} n_l.cknw(c) \leftarrow \begin{cases} k_c', & k_c' \text{ is newer than } k_c \\ k_c, & \text{otherwise} \end{cases} \\ chan(t, n_r, n_l) \leftarrow (k_1, \dots, k_n) \end{array}} \quad \text{(receive)}$$

The rules (send) and (ip-send) describe sending a replica via a channel whose endpoints meet the gossip condition (i.e., they are within wireless communication range, etc.). Here, sending is represented by adding the replica at the end of the channel's queue. Specifically, according to (manet-send), the replica is sent via a MANET communication link only if the pair sender node-replica meets the communication boundary of an ensemble. In case of an IP communication link, as described by (ip-send), both the pair sender-replica and receiver-replica need to meet the communication boundary for the replica to be sent (as explained in Section **Error! Reference source not found.**, technically the receiver-replica pair is to be evaluated by querying a registry). The rule (receive) describes asynchronous reception of a replica from a non-empty channel, represented as removing the first element from the channel's queue. Note that the node's replica is only updated if the received knowledge is newer than the one stored in the replica.

3.5 Real-time aspects

Being completely non-deterministic, the transition system generated by the presented rules can also capture behaviors that are not realistic w.r.t. real execution. In particular, as DEECo targets in general real-time CPS we focus on real-time properties of the transition traces. In principle, we allow only those transition traces that are possible with respect to real-time periodic scheduling of the system processes, ensemble knowledge exchange, and knowledge dissemination. In a way, these restrictions impose a fairness constraint on the transition traces. The technical details on connecting the semantics with time can be found in [2].

4 Conclusions

In this report, we presented a computational model representing a synergy of software component model abstractions and gossip-based communication primitives as a promising solution for engineering scalable dynamic decentralized cyber-physical systems. Our approach relies on providing architecture-level descriptions that feature communication groups (captured by communication boundaries) and allow us "driving" the gossip efficiently. Our current and future work involves experimentation with different gossip algorithms (e.g., a promising direction is to employ location-based algorithms where GPS devices are required).

References

[1] Ali, R. Al, Bures, T., Gerostathopoulos, I., Keznikl, J. and Plasil, F. 2014. Architecture Adaptation Based on Belief Inaccuracy Estimation. *To appear in Proc. of WICSA'14* (Apr. 2014).

[2] Barnat, J., Benes, N., Bures, T., Cerna, I., Keznikl, J. and Plasil, F. 2013. Towards Verification of Ensemble-Based Component Systems. *To appear in Proc. of FACS'13* (Nanchang, China, Oct. 2013).

[3] Beetz, K. and Böhm, W. 2012. Challenges in Engineering for Software-Intensive Embedded Systems. *Model-Based Engineering of Embedded Systems*. Springer. 3–14.

[4] 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.

[5] Friedman, R. and Gavidia, D. 2007. Gossiping on MANETs: the Beauty and the Beast. *ACM SIGOPS Operating Systems Review*. 41, 5 (Oct. 2007), 67–74.

[6] Keznikl, J., Bures, T., Plasil, F. and Kit, M. 2012. Towards Dependable Emergent Ensembles of Components: The DEECo Component Model. *Proc. of WICSA'12* (Aug. 2012), 249–252.

[7] Williams, B. and Camp, T. 2002. Comparison of broadcasting techniques for mobile ad hoc networks. *Proceedings of MobiHoc'02*. (2002), 194.

4.5 Employing Domain Knowledge for Optimizing Component Communication

Michał Kit,
František Plášil,
Vladimír Matěna,
Tomáš Bureš,
Ondrej Kováč

In proceedings of the **18th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE '15)**.

Published by ACM,
pages 59-64,
ISBN 978-1-4503-3471-6,
May 2015.

The original version is available electronically from the publisher's site at http://dx.doi.org/10.1145/2737166.2737172.

Summary of the Paper

The following paper is a continuation of the work done on communication optimization initiated by the idea of *communication boundary* given in Section 4.3. In this publication, the focus is centered around optimizing component data exchange over infrastructure-based networks. The new concept of so-called *communication groups* is proposed to be added to the DEECo ensemble specification, which is then exploited by the jDEECo runtime to improve network utilization by routing component data between those components that belong to a common ensemble. For that purpose, the jDEECo runtime introduces dedicated nodes called *groupers* that are communicated (by the means of the Gossip protocol) by all other nodes in the network and as a response, groupers advise them with their potential peers hosting relevant (in terms of belonging to the same ensemble) components. The example scenario (given in Section 2), on which the idea has been validated, consists of road trains, composed of vehicles capable of automatic driving. The vehicles within a road train need to organize between each other in a similar way as the road train themselves in order to form longer, compound trains when possible. Using the example scenario, the main concepts of the DEECo component model are also described in this section. Following, Section 3 of the paper presents the jDEECo platform specifics corresponding to its heterogeneous (i.e. infrastructure-based and infrastructure-less) deployment. In the infrastructure-based part, the main idea of the communication group is presented on both the modeling level (i.e. the DSL ensemble specification amendments) and the jDEECo platform realization extended by the grouper nodes. Subsequently, the evaluation section (i.e. Section 4) provides the aforementioned extensions to the jDEECo as well as the implementation of the use-case scenario proving the validity of the main concept. Furthermore, scenario simulations performed within the jDEECoSim platform provide measurements showing a considerable gain of the implemented optimization by reduced number of messages exchanged between remotely deployed components. Further, the discussion section elaborates more on the idea of communication groups and its possible implications compared to the communication boundary idea.

Finally, some related work is given in Section 6, which is then followed by the plans on the future work and concluding remarks.

Author Contribution and Goals Addressed

Considering author's contribution in this work, it consists of participation in the final idea formulation, articulating the idea into the paper as well as taking the guiding role in the evaluation section implementation.

In terms of this work correspondence to the goals formulated in Section 1.3, the idea of communication groups targets the goal **G2b** that speaks about component communication improvements.

Employing Domain Knowledge for Optimizing Component Communication

Michal Kit[1], Frantisek Plasil[1], Vladimir Matena[1], Tomas Bures[1,2], Ondrej Kovac[1]

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

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

{kit, plasil, matena, bures, kovac}@d3s.mff.cuni.cz

ABSTRACT

The emerging area of (smart) Cyber Physical Systems (sCPS) triggers demand for new methods of design, development, and deployment of architecturally dynamic distributed systems. Current approaches (e.g. Component-Based Software Engineering and Agent-Based Development) become insufficient since they fail in addressing challenges specific to sCPS such as mobility, heterogeneous and unreliable deployment infrastructure, and architectural dynamicity. The strong dependence on the underlying communication infrastructure, often combining ad-hoc established links typical for wireless connectivity with more reliable connections of infrastructural networks, requires a novel method to optimize system deployment. In this paper we propose such a method based on the domain knowledge elicited from design level specification. As a proof of concept, we have provided an extension to the DEECo (Dependable Emergent Ensembles of Components) model and validated it on a scenario from the domain of Vehicular Area Networks.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems – d*istributed applications*; D.2.10 [**Software Engineering**]: Design; D.2.11 [**Software Engineering**]: Software Architectures;

Keywords

Cyber-physical systems; Domain knowledge; Component communication

1. INTRODUCTION

Recent growth in connectivity of electronic devices results in the birth of new kind of distributed systems regarded often as the Internet of Things (IoT) or as (smart) Cyber Physical Systems (sCPS) [1]. There are already multiple examples of such systems, stemming from different usage domains such as assisted living, intelligent transportation, and (mobile and ad-hoc) cloud computation. They are usually composed of autonomous components designed to execute independently in order to support resilience of the system. Nevertheless, components need to communicate with each other, exchanging data that allows them to perform cooperative actions. The way components interact together along with the conditions under which their interaction occurs are usually grasped by an architecture description articulated during the design phase.

CBSE'15, May 04-08, 2015, Montreal, QC, Canada
Copyright 2015 ACM 978-1-4503-3471-6/15/05…$15.00
http://dx.doi.org/10.1145/2737166.2737172

After being implemented, components are deployed to physical nodes interconnected by the means of ad-hoc and/or infrastructure networks, each of these requiring a dedicated approach towards data dissemination.

Problem statement: Depending on its scale, deployment of sCPS may involve a large number of physical nodes with different communication infrastructure. To optimize utilization of a network, some information about the communication aspects should be introduced yet at the design level when the application architecture is decided. This, however may violate infrastructural transparency needed to allow for deployment independency. In the end the challenge here is to find a solution that would sustain communication transparency at the architectural level and, at the same time, allow for its optimization at the infrastructure level.

Goal: This paper aims to propose a method targeting the problem above and to show its feasibility on a case study implemented in the existing DEECo (Dependable Ensembles of Components) [2] component model and framework. Specifically, we address the problem by introducing communication groups based on adding domain-specific knowledge to the architecture; this allows optimization of the network use while preserving the level of abstraction typical for architectural design.

The rest of the paper is structured as follows: Section 2 describes motivating example and overviews the background technology – DEECo. Section 3 introduces the idea of using domain specific knowledge for communication optimization and in Section 4 the benefits of the idea are evaluated. Section 5 discusses potential applications of the communication groups. Section 6 focuses on related work while Section 7 concludes the paper by summarizing its contribution.

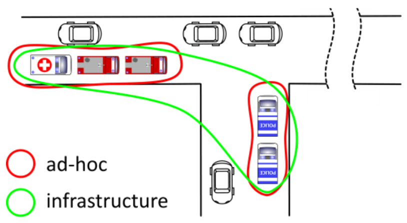


Figure 1: Road trains scenario – ad-hoc and infrastructure networks employment

2. MOTIVATION AND BACKGROUND

2.1 Road - trains scenario

As a running example, consider a scenario of emergency vehicles forming *road-trains* (a chain of vehicles heading towards the same destination). The purpose of a road train is to optimize movement (in terms of speed, safety, and traffic disruption) of emergency vehicles towards the site of an accident. We assume that each vehicle is equipped with the necessary hardware enabling both short-range wireless communication (via MANETs) as well as infrastructure-based connectivity (long range, dedicated to emergency services). Vehicles within a single road train communicate in order to maintain proper internal organization of the train and to ensure satisfiability of the safety requirements such as minimal distance between vehicle, maximal speed etc. Furthermore, all vehicles (also across different road trains) exchange information necessary to form a road train, including desired destination, current location etc. The scenario together with two types of data flows is illustrated in Figure 1.

In this scenario, we focus on the organization of vehicles seen as autonomous components that need to communicate globally (to form a road train) and locally (while in a road-train). Whereas the local coordination requires low latency in data exchange, which is achieved by short-range communication, global coordination accounts for optimality in terms of network utilization.

2.2 Background: DEECo

Proposed for development of dynamic CPS, the DEECo (Dependable Emergent Ensembles of Components) component model and its framework was introduced [2] for designing applications of autonomous components and their dynamic ad-hoc groups (*ensembles*) that the components establish serving for their communication.

A DEECo component is an independent unit of computation and deployment. In the scenario, components correspond to the main actors of the system (i.e. vehicles). The template of these components is specified in Figure 2 by the Vehicle specification. Its state is captured by *knowledge* (a set of attributes - lines 7-14) and operational functionality by *processes* (lines 15-20). Every component features a number of *roles*, i.e., sets of knowledge fields (lines 1-4, 6), which provide contract between the component and ensembles. Processes are executed by the runtime periodically or in a triggered manner. Line 19 demonstrates a specification of periodic execution of the processes with a given time period. Each process execution starts with atomic reading (a part of) of component knowledge, executing the process body, and ends with atomic writing the results back to the knowledge.

In Figure 2, ensembles reflect the two types of communication groups of vehicles - within a road-train and across all the vehicles heading to the same destination. For instance, consider the SameDestination ensemble definition (lines 22-34). The goal here is to propagate information about the vehicle's desired destination to other vehicles so that they can coordinate movement to form road-trains. The figure illustrates that an ensemble definition in DEECo contains a *membership condition* specifying which components can join the ensemble (lines 25-28), and a prescription of *knowledge exchange* between its *coordinator* and *members* (lines 29-33). The coordinator and potential members are characterized by specific roles (lines 23-24). An ensemble is instantiated and dissolved by the DEECo runtime framework, which periodically (line 34) checks the membership condition. Whenever the ensemble is formed (i.e. there is a coordinator and at least one member), the runtime framework periodically performs

```
1.  role OtherVehiclesAgregator:
2.    otherVehicles, position
3.  role LocalTrainAgregator:
4.    trainId, localVehicles, speed, position
5.
6.  component Vehicle features OtherVehiclesAgregator, LocalTrainAgregator
7.     knowledge:
8.        ID = V1
9.        otherVehicles = [(V2, {50.0722, 14.4568}), (V4, {50.0745, 14.2356})]
10.       localVehicles = [(V3, {50.25636, 14,4568}, 45.6)]
11.       position = {50.075306, 14.426948}
12.       speed = 54.2
13.       destination = 109
14.       trainId = 5
15.     process measureSpeed
16.          out speed
17.       function:
18.          speed ← SpeedSensor.read()
19.       scheduling: periodic( 500ms )
20.    … /* other process definitions */
21.
22. ensemble SameDestination:
23.    coordinator: OtherVehiclesAgregator
24.    member: OtherVehiclesAgregator
25.   membership:
26.       member.destination.Address = coordinator.destination.Address
27.       AND !member.isRoadTrainMember
28.       AND !coordinator.isRoadTrainMember
29.    knowledge exchange:
30.       coordinator.otherVehicles ← { (m.ID, m.position) | m ∈ members }
31.       for(m ∈ members)
32.          m.otherVehicles ←{ t ∈ coordinator.otherVehicles | t.ID ≠ m.ID }
33.          m.otherVehicles ← (coordinator.ID, coordinator.positon)
34.    scheduling: periodic( 700ms )
35.
36. ensemble TrainManagement:
37.    coordinator: LocalTrainAgregator
38.    member: LocalTrainAgregator
39.    membership:
40.       member.trainId = coordinator.trainId
41.    knowledge exchange:
42.       coordinator.localVehicles ← { (m.ID, m.position, m.speed) | m ∈ members }
43.       for(m ∈ members)
44.          m.localVehicles ←{ t ∈ coordinator.localVehicles | t.ID ≠ m.ID }
45.          m.localVehicles ← (coordinator.ID,
46.                             coordinator.position,
47.                             coordinator.speed)
48.    communication boundary:
49.          (sender: LocalTrainAgregator, node: NodeKnowledge) =>
50.              ∃ component ∈ node.components:
51.                  hasRole(component, LocalTrainAgregator) AND
52.                  component.knowledge.trainId = sender.trainId
53.    scheduling: periodic( 200ms )
```

Figure 2: Examples of DEECo component and ensembles of the road trains scenario

the knowledge exchange by transferring data form the members to the coordinator (and vice versa) as specified by the mapping in the ensemble definition. A component specification may feature multiple roles; consequently, a component may be a member/coordinator of many ensembles at a time.

It should be emphasized that, knowledge exchange, realized by the ensembles to which a particular component belongs, is the only means of inter-component communication.

3. COMMUNICATION EMPLOYING DOMAIN KNOWLEDGE

Tailored for development of sCPS, the DEECo component model allows designing a system at the architecture level without considering aspects related to its actual deployment - component distribution, communication infrastructure, and even its scaling in terms of the eventual number of component instances. Such an abstraction level simplifies modeling and development of the system, as it allows reasoning about components and ensembles in isolation, a crucial property when dealing with complex systems. Problems arise when it comes to deployment of the system, since there is a gap between the abstraction level of design and runtime infrastructure. This typically implies the need to apply standard generic methods for communication among distributed nodes. In particular in sCPS the efficiency of communication can be substantially improved by employing application domain data to optimize the deployment of the system.

In this section, we present how this can be achieved in DEECo by employing the concept of communication boundary [3], and, as a key contribution, the novel idea of communication groups.

3.1 Ad-hoc Networks

DEECo and specifically its Java realization jDEECo [4], supports ad-hoc communication via MANETS. It relies on periodic channel-level broadcasts (rebroadcasts) of component knowledge. In a system, this allows a node not only be aware of the knowledge of the components it hosts but also to *learn* about knowledge of other (remote) components. This approach is appropriate for MANETS that are not fully reliable and prone to frequent disconnections due to radio interference and mobility of nodes.

The communication protocol in jDEECo is based on *bounded gossiping* [3], where components' knowledge rebroadcasting is limited by *communication boundaries* articulated in ensemble specifications. A communication boundary is employed by a node for deciding whether or not to rebroadcast the component knowledge heard from other nodes. This way, by constraining component knowledge dissemination to a specific geographical area, this mechanism allows better utilization of the communication channel, which in wireless settings comes at a great price.

An example of communication boundary is given in Figure 2 (lines 48-52) when the component knowledge data dissemination is bounded to the nodes of the vehicles participating in the same road train.

The specification of a communication boundary, is given as a predicate formulated on the component knowledge to be rebroadcasted and the knowledge of a rebroadcasting node. This way, the communication boundary reflects only the application domain-specific knowledge known at the architectural level. (Specifically, no information about future deployment is required.) In case of Figure 2, the domain-specific knowledge captures the fact that a road-train is a spatially connected structure and thus it is sufficient to involve only the road-train nodes in the rebroadcasting. As an aside, this is the only enhancement to the original semantics of the ensemble as specified at the architectural level.

3.2 Infrastructure Networks

The benefits of the communication boundary is apparent for ad-hoc networks; nevertheless the idea is also applicable when dealing with more reliable infrastructure networks (IN for short). In such settings, however, one can do more than just restrict data retransmissions. Having a topology that does not change often (in particular if established links hold for a relatively long time), a routing mechanism can be introduced to provide for optimality with respect to, e.g., bandwidth utilization, latency, and computation balancing.

Therefore, jDEECo utilizes gossiping in case of infrastructure networks [5]. As a data dissemination protocol, gossiping is resilient to communication failures. Nevertheless, depending on the application, standard gossiping may still be costly, especially in terms of the amount of data being transmitted. Specifically, in jDEECo standard gossiping causes that component knowledge is published periodically to all nodes in a system, which does not scale well for large-scale systems.

Communication groups: To mitigate the problem of unnecessary data transmission over the network, we propose an extension to ensemble definition by introducing *communication group*, delineated according to the component knowledge of the coordinator and members of an ensemble. The basic idea is to introduce the groups of components (members) that are related to each other in terms of a specific knowledge value (e.g. having the same value of the destination attribute). Such a group serves as a hint for optimizing deployment in terms of communication efficiency by restricting and localizing the area in which discovery of components to form an ensemble is performed. Defined again at the architecture level via component knowledge specified in roles, orthogonally to the membership, knowledge exchange, and communication boundary, the concept of communication group just enhances the original semantics of ensemble, not modifying the meaning of other DEECo abstractions. For illustration, consider line 4 in Figure 3, indicating that communication groups will be based on the destination value in the coordinator's knowledge. In this case vehicles going to the same destination (expressed by the membership condition) compose communication groups, each of them corresponding to a specific value of the destination attribute in the coordinator's knowledge. The situation is visualized in Figure 4, where ensembles of different emergency vehicles trains are heading to distinct locations in Prague 6 and in Prague 4 districts.

Groupers: Communication groups are utilized to optimize deployment, where they support the planning of inter-node communication links. For that reason an extension to the jDEECo runtime environment is proposed by introducing the concept of *grouper*. The basic idea is that a grouper limits the gossiping only to the nodes that host the components belonging to a particular communication group. Thus a grouper employs the communication group specification. Technically a grouper enhances the jDEECo runtime environment in the following way. The environment contains a set of jDEECo runtime instances (Figure 5). Each of them hosts a set of components, the knowledge of which is gossiped around, using the addresses of other nodes stored in *its recipient table*. In the enhancement, a grouper can also be referenced in the recipient table as illustrated in Figure 5. It is assumed that a grouper (i) is a representative of a communication group(s), (ii) is equipped with all the ensemble definitions in the system, (iii) has access to knowledge of all components needed to evaluate the membership

```
1.   ensemble SameDestination:
2.     coordinator: OtherVehiclesAgregator
3.     member: OtherVehiclesAgregator
4.     communication group: coordinator.destination.CityDistrict
5.     membership:
6.          member.destination.Address = coordinator.destination.Address
7.          AND !member.isRoadTrainMember
8.          AND !coordinator.isRoadTrainMember
9.     knowledge exchange:
10.         coordinator.otherVehicles ← { (m.ID, m.position) | m ∈ members }
11.         for(m ∈ members)
12.              m.otherVehicles ←{ t ∈ coordinator.otherVehicles | t.ID ≠ m.ID }
13.              m.otherVehicles ← (coordinator.ID, coordinator.positon)
14.    scheduling: periodic( 700ms )
```

Figure 3: An example of communication group specification

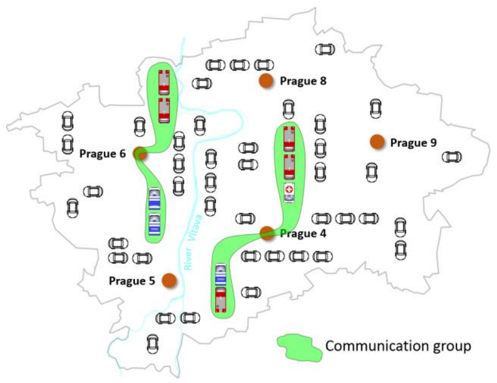


Figure 4: Illustration of communication groups; each is associated with an instance of SameDestination

conditions, and (iv) can modify the recipient tables that contain a reference to it. The basic functionality of a grouper is to continuously monitor the current ensemble memberships of all the components in the groups it represents and update the recipient tables accordingly. By modifying the recipient tables, a grouper implicitly routes the component knowledge to the most relevant nodes (i.e. hosting the same ensemble components) in the network. Figure 5 exemplifies the whole idea on the SameDestination ensemble, the communication group of which depends on the destination field in the coordinator's knowledge. In this case the grouper dedicated to Prague 6 continuously monitors the ensemble membership status of all the components it is aware of. If necessary, it updates the recipient tables of the respective jDEECo runtime instances on the nodes hosting the components in the SameDestination ensemble. The specification of the communication group allows a node to determine the groupers for a given pair of an ensemble type and a component hosted on the node. This way the knowledge of the component is routed only to a limited set of groupers (as opposed to being propagated throughout the whole system). Also, as communication groups are

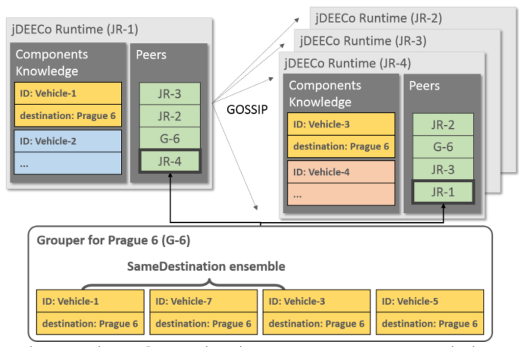


Figure 5: jDEECo runtime instances – groupers associations

typically geographically or network-wise localized, they lead to a decentralized solution, potentially characterized by low-latency. The decentralization also means that the operation is possible even in case that the infrastructure network gets partitioned into a number of disconnected subnets (i.e. without global internet connectivity).

4. EVALUATION

4.1 Proof of concept

As a proof of concept we have simulated the road trains scenario and conducted several experiments, allowing us to assess the applicability of the method[1]. We used the total number of messages exchanged in the system as a metric for expressing communication efficiency. The simulation, conducted with use of MATSim [6], was focused on optimization of emergency vehicles' routings across realistic road network of the Prague city provided by OpenStreetMap [7]. Firefighter, police, and ambulance vehicles were considered as the emergency vehicle types. The locations of ambulance, police and firefighter bases were set according to their real locations. For simplicity, all non-road objects and several minor roads were removed from the original map which yielded a road network covering the area of approximately 100km^2.

The simulation comprises three groups of experiments: (i) emergency call response by 3 vehicles, (ii) emergency call response by 5 vehicles, and (iii) single large road train (convoy with the right of the way). The groups (i) and (ii) encompass experiments differentiated by number of concurrent emergency calls (1, 2, 3, 5, 10, 15, 20), while (iii) encompasses experiments with several road train sizes (3, 5, 10, 15, 20).

As to (i) and (ii), when an emergency call is issued (e.g. a serious car crash), vehicles are dispatched to the accident site (destination). In the simulation, the emergency vehicles heading to the same destination aim at forming a road-train to make it easier to clear their path in heavy traffic by driving closely behind each other. The emergency vehicles are dispatched from the emergency service bases as close to the destination as possible. Specifically it is assumed that: in (i) one of each emergency vehicle type is sent to every destination, in (ii) two ambulance, two firefighter and one police vehicles are sent to every destination, in (iii) emergency vehicle types are not distinguished.

Once a vehicle is on its way to the accident site, it aims at following another emergency vehicle heading to the same destination. A road train is established, when the distance between two solo vehicles heading the same destination is negligible in a street. A vehicle is allowed to join a road train only when its prolongation of its route to the destination is minor and has the ability to increase its speed temporarily.

In order to show that the results do not depend on particular routing and destination choice, 10 different simulation runs parametrized by destination choice were executed.

4.2 Experiment results and lessons learned

In (i) and (ii) a key result of these sets of experiments is the proof of communication complexity reduction (from quadratic to linear). Recall that this complexity metric is the number of IN messages – in our case those were the IP messages. For the group (i) the number of IP messages was measured (Figure 6); for the group (ii) this

[1] Source code of the scenario implementation used in the experiments is available at: http://github.com/d3scomp/cbse-2015-tutorial

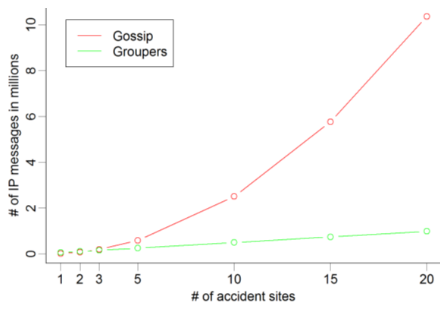


Figure 6: Communication complexity comparison of gossip and groupers; experiment group (i) – 3 vehicles per accident

measurements are in (Figure 8). The reason for not considering MANET messages is that these are local and thus not influencing the infrastructure network load, even though small fraction of these is inherently rebroadcasted in MANET network. From these figures, it follows that when just gossip is applied, the number of IP messages grows quadratically with the number of vehicles. This is caused by the fact that IP messages from a vehicle are sent to all of other vehicles. On the contrary, when groupers are applied the IP messages are sent only to the vehicles sharing a particular destination, the number of IP messages is linear in number of groups while assuming the size of the group is constant. Moreover, here it is also visible that the effect of improvement starts at a minimal number of destinations (such as 3 in Figure 6), since there is an overhead of communication among groupers.

Note that a system that enables message passing between the MANET and IN networks (such as jDEECo originally) needs to be configured in such a way that messages from different communication groups do not leak from one communication group to another, otherwise this would harm the positive effect of communication groups. As an aside, in this simulation this was ensured by preventing rebroadcasting of IP messages by MANET.

Finally, domain specific knowledge can be further exploited by evaluating ensemble membership conditions in groupers. Such a feature would enable distribution of knowledge only to those nodes that host components satisfying a particular ensemble membership condition. In the road trains scenario (Figure 3), a vehicle that is a member of a road train is not a member of an instance of SameDestination any more, thus not being subject to the respective knowledge exchange, since only the "solo" vehicles and road-train leaders need to communicate via IN network. Therefore, thanks to

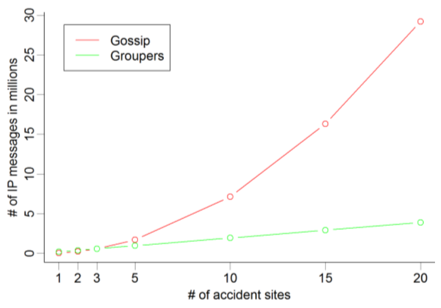


Figure 8: Communication complexity comparison of gossip and groupers; experiment group (ii) – 5 vehicles per accident

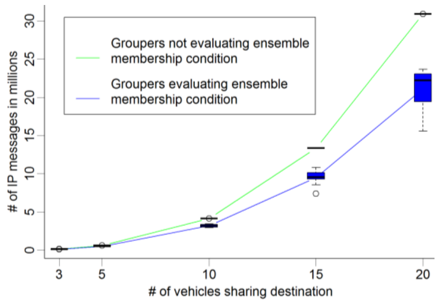


Figure 7: The effect of introducing groupers evaluating ensemble membership condition; experiment group (iii) – single large road train

ensemble membership condition evaluation in groupers, it is possible to exclude those vehicles from communication group. The effect of this optimization would be minimal in experiment groups (i) and (ii), since the road-trains considered are relatively short. In order to study this effect, the group (iii) was introduced. From Figure 7 it is clearly visible that introducing groupers evaluating ensemble membership condition further reduces the number of IP messages for larger road trains. (Note that due to a higher variance of results in Figure 7, we use box-plots instead of simple points. Results in Figure 6 and 7 exhibit very low variance; thus we show only the mean values.)

5. DISCUSSION

The idea of the communication groups is applicable in most sCPS, nevertheless it shows its full strength when a combination of both ad-hoc and infrastructure networks takes place. Advantageously, it accounts for infrastructural dynamicity and mobility of network nodes in particular. In the scenario described in Section 2, the communication groups are for simplicity static in the sense that when a vehicle is assigned its destination, it is rather unlikely that this will be modified. However, the approach equally supports situations where a communication group depends on a dynamically changing factor such as current position of a vehicle. For instance, it is possible to specify the group in such a coarse way that the current position of the vehicle is refers just a particular Prague districts. Then, depending on the current position of the vehicle, the jDEECo runtime instance - grouper associations would vary over time, optimizing the network traffic (with respect to data latency and amount of data sent) via the nearest grouper in the district.

While communication boundary constrains the actual communication topology, communication groups provide for (context-aware) routing mechanisms. These two concepts complement each other, and in case of infrastructure networks can be used either separately or in combination, effectively providing for different scenarios of component knowledge dissemination. Communication group offers more flexibility in terms of the possible optimization strategies to be implemented during deployment process. Depending on non-functional requirements, deployment optimizing network traffic in terms of data latency can be achieved. In such cases, it is desirable to bring groupers as close as possible (considering geographical distance) to the relevant nodes. On the other hand, if the main concern is balancing (or optimizing) the utilization of computational resources, then the deployment would be based mainly on their availability. Such flexibility is useful in heterogeneous deployments, where a part of the network is to be latency sensitive and another latency tolerant.

As an example consider imposing additional requirement to the scenario from Section 2 that intra-train communication should be latency sensitive and the communication between road trains latency tolerant.

6. RELATED WORK

The idea of communication groups relates to Distributed Hash-Tables (DHT) which introduce key-space partitioning [8], [9] and overlay networks [10]. The former assign a range of keys to particular network nodes that take the responsibility for storing the actual value corresponding to a key. The latter store references to other nodes to allow each node to query another node during key lookups. In this context communication groups can be partially interpreted also as a key-value storage problem, where the key is the particular value of the communication group specification given in the ensemble DSL definition, while the value is the set of components forming the ensemble. Other commonalities of communication groups and DHT include: implementation transparency, topological dynamism and redundancy mechanisms (increasing the overall reliability of the system). All in all, DHT may serve as supporting technology for implementing the part of communication group functionality that provides a mapping between a particular group and a set of network nodes. The main difference thus lies in the level of abstraction, where communication groups are primarily an architectural concept, while DHTs belong to middleware.

In terms of benefits of communication group, they go along the same lines as fog computing (also edge computing). Fog computing extends the concept of cloud computing by pushing the data and computation from centralized nodes (data centers) closer to end devices – i.e. to the edge of the network [11], [12]. Similar to cloud, fog provides data, computation power, and networking services more likely in a latency-free manner. Basically, communication group resembles the idea of fog computing of coping with demand for low latency but does it by different means (employing domain knowledge in particular).

With respect to exploiting domain-specific knowledge to improve network utilization communication group resembles context-aware routing protocols [13], a technique used in wireless (mesh) networks or delay-tolerant mobile ad hoc networks. It uses various information from the environment (context) to discover optimal path from the source to a destination or to adapt to changes in network topology. In [14], the authors propose a method building on node mobility and the history of establishing links with other nodes including their location. In a similar vein, geographic routing (or geo-routing) [15] relies on the geographical position of nodes. In addition to classical packet addressing, it also employs indication of the actual geographic position of a target node. The concept of communication groups takes the idea of context-aware routing a step further. Driven by application domain knowledge, it is more flexible with respect to the information being exploited in order to provide for more accurate addressing. Even though context-aware routing is fully distributed, communication groups are dedicated to infrastructure networks (see Section 3.2), where communication links are relatively stable and reliable, making the centralization aspect of groupers not a big issue.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented a DEECo dedicated method that introduces communication groups exploiting application domain knowledge in order to optimize communication infrastructure utilization. As an extension to the DEECo model, communication groups increse efficiency of sCPS with respect to their deployment. The method relies on providing architecture-level descriptions that define communication groups and allow component knowledge routing according to custom preferences (latency sensitivity, resource utilization). Our current and future work involves adding features such as key partitioning and improved data exchange to groupers. Further, we plan to apply OMNet++ based simulations to obtain measurements reflecting network latency.

8. ACKNOWLEDGMENTS

This work was partially supported by the EU project ASCENS 257414 and by Charles University institutional funding SVV-2015. 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.

9. REFERENCES

[1] P. Barsocchi, S. Chessa, I. Martinovic, and G. Oligeri, “A cyber-physical approach to secret key generation in smart environments,” *J. Ambient Intell. Humaniz. Comput.*, vol. 4, no. 1, pp. 1–16, 2013.

[2] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil, “DEECo – an Ensemble-Based Component System,” in *Proc. of CBSE’13*, 2013, pp. 81–90.

[3] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil, “Gossiping Components for Cyber-Physical Systems,” in *Software Architecture*, vol. 8627, P. Avgeriou and U. Zdun, Eds. Springer International Publishing, 2014, pp. 250–266.

[4] jDEECo [Online]: https://github.com/d3scomp/JDEECo.

[5] S. Voulgaris, M. Jelasity, and M. van Steen, “A Robust and Scalable Peer-to-peer Gossiping Protocol,” in *Proc. of 2nd AP2PC*, Berlin, Heidelberg, 2004, pp. 47–58.

[6] MATSim [Online]: http://www.matsim.org/.

[7] OpenStreetMap [Online]: http://www.openstreetmap.org.

[8] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, “Looking Up Data in P2P Systems,” *Commun ACM*, vol. 46, no. 2, pp. 43–48, Feb. 2003.

[9] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web,” New York, NY, USA, 1997, pp. 654–663.

[10] K. Dhara, Y. Guo, M. Kolberg, and X. Wu, “Overview of Structured Peer-to-Peer Overlay Algorithms,” in *Handbook of Peer-to-Peer Networking*, X. Shen, H. Yu, J. Buford, and M. Akon, Eds. Springer US, 2010, pp. 223–256.

[11] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog Computing and Its Role in the Internet of Things,” New York, NY, USA, 2012, pp. 13–16.

[12] L. M. Vaquero and L. Rodero-Merino, “Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing,” *SIGCOMM Comput Commun Rev,* vol. 44, no. 5, pp. 27–32, Oct. 2014.

[13] C. Boldrini, M. Conti, and A. Passarella, “Social-based autonomic routing in opportunistic networks,” in *Autonomic Communication*, A. V. Vasilakos, M. Parashar, S. Karnouskos, and W. Pedrycz, Eds. Springer US, 2009, pp. 31–67.

[14] Y. Yang, J. Wang, and R. Kravets, “Designing routing metrics for mesh networks,” in *In WiMesh*, 2005.

[15] M. Musolesi and C. Mascolo, “CAR: Context-Aware Adaptive Routing for Delay-Tolerant Mobile Networks,” *IEEE Trans. Mob. Comput.*, vol. 8, no. 2, pp. 246–260, Feb. 2009.

4.6 An Architecture Framework for Experimentations with Self-Adaptive Cyber-Physical Systems

Michał Kit,
Ilias Gerostathopoulos,
Tomáš Bureš,
Petr Hnětynka,
František Plášil

In proceedings of the **10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '15)**.

Published by IEEE,
pages 93-96,
May 2015.

The original version is available electronically from the publisher's site at http://dx.doi.org/10.1109/SEAMS.2015.28.

Summary of the Paper

The following publication is an artifact paper that introduces the DEECo framework as a comprehensive solution for building self-adaptive cyber-physical systems. The framework, with its tooling, addresses each step of the system development cycle: (i) the system design with the IRM method, (ii) development with the jDEECo platform and finally (iii) the verification with the jDEECoSim tool. The whole explanation of the framework is based on a simple use-case example (given in Section II), which consists of collaborative vehicles working together to optimize their parking procedure. The main idea behind is that vehicles exchange locally gathered (via sensors) information and build a partial view on the system current situation with respect to parking spot occupancy. This allows them to make more accurate decisions on where to park. In Section III, the DEECo component model is introduced, describing its basic concepts (i.e. components and ensembles) and their usability in the context of modeling self-adaptive systems. Furthermore, a brief note on the component and ensemble development process in the jDEECo is given depicting some of the example code snippets. This is followed by Section IV, which informs a reader about the jDEECoSim tool and its main features. Finally, the IRM method is presented as a goal-based design technique for modeling self-adaptive systems built over the DEECo components and ensembles.

Author Contribution and Goals Addressed

The paper is a compilation of different tools/methods researched within the group and as such, the author's contribution in this context goes mainly to sections related to author's work. This includes the DEECo component model together with the jDEECo runtime environment part (Section III) as well as the description of the jDEECoSim tool (Section IV).

In terms of goals addressed by this publication, the article is a collection of different aspects of the DEECo framework (including the DEECo model, jDEECo platform and jDEECoSim tool) and as such contributes to all three goals (**G1**, **G2**, and **G3**). As stated earlier, the focus of the article is to present the entire DEECo framework as a method for facilitating incorporation of self-adaptive techniques into a developed system. Self-adaptation is one of the concerns of SCPS (see Section 1), and even though it is not explicitly formulated as a research goal can be also considered as partially addressed by this work.

An Architecture Framework for Experimentations with Self-Adaptive Cyber-Physical Systems

Michal Kit, Ilias Gerostathopoulos, Tomas Bures, Petr Hnetynka, and Frantisek Plasil
Charles University in Prague
Faculty of Mathematics and Physics
Department of Distributed and Dependable Systems
Prague, Czech Republic
{kit, iliasg, bures, hnetynka, plasil}@d3s.mff.cuni.cz

***Abstract*—Recent advances in embedded devices capabilities and wireless networks paved the way for creating ubiquitous Cyber-Physical Systems (CPS) grafted with self-configuring and self-adaptive capabilities. As these systems need to strike a balance between dependability, open-endedness and adaptability, and operate in dynamic and opportunistic environments, their design and development is particularly challenging. We take an architecture-based approach to this problem and advocate the use of component-based abstractions and related machinery to engineer self-adaptive CPS. Our approach is structured around DEECo – a component framework that introduces the concept of component ensembles to deal with the dynamicity of CPS at the middleware level. DEECo provides the architecture abstractions of autonomous components and component ensembles on top of which different adaptation techniques can be deployed. This makes DEECo a vehicle for seamless experiments with self-adaptive systems where the physical distribution and mobility of nodes, and the limited data availability play an important role.**

***Index Terms*—Component framework, self-adaptation, cyber-physical systems**

I. Introduction

Adaptation to different situations and environments has become a common necessity in smart Cyber-Physical Systems (CPS) [1] – i.e., systems closely interacting with physical world entities. Such systems are typically open-ended and have to be capable of supporting new requirements and needs. At the same time, these systems are deployed in heterogeneous and ever-changing (sometimes even hostile) environments and thus have to promptly react to these changes. Due to limited connectivity, smart CPS typically have to perform adaptation in a decentralized manner, which adds to the overall complexity of designing the adaptation. Moreover, in complex enough systems such as modern smart CPS, the mutual dependencies of different *local* adaptation strategies may have an unexpected *global* impact – a behavior often referred to as *emergent*.

To correctly design complex self-adaptive smart CPS is thus a challenging task, which is only partially addressed by existing software engineering models and approaches. This stems from the fact that a correct design of smart CPS has to apply a holistic view that takes into account the overall system goals, the operational models of the system and its environment (along with the uncertainty present in these models), and the employed communication models (along with issues related to latencies and communication unavailability).

In this paper, we present DEECo (Dependable Emergent Ensembles of Components) [2] – a model and framework for developing complex smart CPS. In its model, DEECo provides the holistic view that combines the goals of a system, the system's operational model (including real-time constraints), and realistic communication model (including limited communication and latencies). With its framework, DEECo allows large-scale simulations of complex CPS. Combined with the real-time perspective of DEECo and the network-accurate simulation of communication, DEECo offers accurate insight into the effects of adaptation strategies in complex smart CPS.

The structure of the paper is as follows: Section II presents the running example of self-adaptive vehicles. Section III discusses the DEECo component model, while Section IV presents its reification delivered by the JDEECo simulation framework. Section V presents the IRM design method used in DEECo, while Section VI concludes the paper.

II. Running Example

To illustrate the DEECo models and significant features, we rely in this paper on a smart parking scenario. In the frame of this scenario, vehicles are equipped with vehicle-to-vehicle (V2V) communication and smart sensors to detect available parking spaces along the streets and exchange their knowledge about the available parking capacity (Figure 1).

From the architectural perspective, vehicles are represented as autonomous components, each consisting of a belief and real-time processes. While the belief (*knowledge* in DEECo) reflects the components' perspective about the available parking spaces, the real-time processes take care of sensing the current position, free parking spaces in visible range, etc. In addition to direct sensing, component enrich their belief by exchanging the belief with other components – i.e., the information about the available parking capacity is exchanged between vehicles that are in proximity (typically via limited-range V2V communication).

This information allows vehicles to adjust their route to effectively find a parking space. Also, the inter-component communication may be used to negotiate with other vehicles for selecting and reserving a parking space.

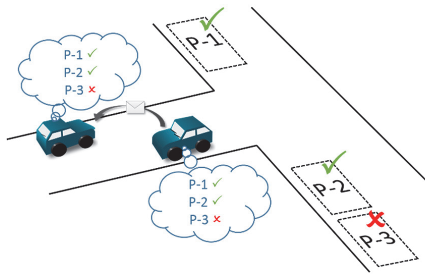


Figure 1: Vehicles sharing data about parking space capacity.

III. DEECO COMPONENT MODEL

DEECo offers straightforward support for development of self-adaptive CPS thanks to the following aspects (structured in sub-sections) that it offers to system architects and developers.

A. Dynamic ensemble-based component model

DEECo is built on the concepts of autonomous *components* and *ensembles* [3]. A component is an independent unit of computation and deployment, while an ensemble is a group of components cooperating to achieve a particular goal.

Ensembles are established/disbanded dynamically at runtime depending on the state of the environment and the state of the components. They can overlap, reflecting the fact that a component may take on multiple roles and pursue multiple goals at the same time (e.g., the goal of having up-to-date information about parking spaces and the goal of making sure that selected parking space is reserved).

The concept of ensembles thus allows forming dynamic component architectures and provides a straightforward reflection of operational goals in the application architecture. The concept of ensembles is further backed by theoretical research in coordination logics [4], which makes it possible to apply related results from statistical model-checking [5] (outside the scope of the artifact presented in this paper).

Figure 2 illustrates the concepts of DEECo's autonomous components and ensembles. It shows the specification (in DEECo DSL) of the smart parking scenario. A vehicle is captured as a Vehicle component – subject to multiple instantiations. It consists of its *knowledge* (i.e., the state of the component and its belief about other components – lines 5-11), and of real-time *processes* (lines 12-16). Processes in DEECo are periodic (time-triggered and event-triggered). This makes it easy to implement both real-time CPS control logic and adaptation logic as MAPE-K [6] loops.

Communication between components is not direct but happens via knowledge exchange between components in an ensemble. Figure 2 shows the CapacityExchangeEnsemble, which reflects the goal of vehicles having up-to-date information about the available parking spaces.

Technically, an ensemble is defined by its *membership* condition and its *knowledge exchange*. Membership determines which components to involve in an ensemble (e.g., all vehicles in proximity), whereas knowledge exchange specifies which knowledge should be exchanged among these components. An ensemble may be instantiated multiple times if the situation described by membership occurs at different places (potentially involving different components).

```
1.  role LinkCapacityAggregator:
2.    linkCapacities, position
3.
4.  component Vehicle features LinkCapacityAggregator
5.      knowledge:
6.          ID = V1
7.          linkCapacities = [(Link_21, 1), (Link_21, 2), …]
8.          position = {50.075306, 14.426948}
9.          speed = 54.2
10.         destination = Link_126
11.         selectedParking = P23
12.      process measureSpeed
13.            out speed
14.         function:
15.            speed ← SpeedSensor.read()
16.         scheduling: periodic( 500ms )
17.     … /* other process definitions */
18.
19. ensemble CapacityExchangeEnsemble:
20.     coordinator: LinkCapacityAggregator
21.     member: LinkCapacityAggregator
22.   membership:
23.     distance(member.position, coordinator.position) < ENSEMBLE_RADIUS
24.   knowledge exchange:
25.         coordinator.linkCapacities ← { m.linkCapacities | m ∈ members }
26.         for(m ∈ members)
27.             m.linkCapacities ←{ coordinator.linkCapacities }
28.   scheduling: periodic( 1000ms )
```

Figure 2: Examples of a DEECo component and an ensemble.

To ease the structural specification of an ensemble, DEECo features two principal roles – ensemble *coordinator* and ensemble *member*. Membership is then expressed as a condition over the knowledge of the coordinator and the knowledge of the members (lines 22-23). Similarly, knowledge exchange is specified as assignment from the knowledge of members to the knowledge of the coordinator and vice-versa (lines 24-27).

The contract between ensembles and components is carried by component *roles* (lines 1-2). The role specifies the knowledge of a component that an ensemble can assume.

An ensemble periodically (in real-time manner – line 28) evaluates the membership condition and executes the knowledge exchange. In evaluating the ensembles, the model takes into account network latencies, which means that the knowledge of a component is available to other components only after certain random time, which is further correlated with the intensity of other network traffic and the geographical distance (in case of multi-hop communication in V2V networks).

B. Openness and extensibility for adaptation strategies

DEECo is open to deployment of different adaptation algorithms or strategies. They can be implemented as component processes and seamlessly integrated with a DEECo application. They can be also dynamically switched in response to: (i) values that are directly sensed by a component (e.g., free parking spaces around a vehicle), (ii) the state of a component (e.g., destination where a vehicle wants to park), and (iii) the belief about the knowledge of other components, including their perception of the environment (e.g., free parking spaces in another street).

```
@Component
public class VehicleComponent {
   public String id;
   public Map<…> linkCapacities;
   public Coord position;
   public Double speed;
   public Link destination;
   public Parking selectedParking;

   @Process
   @PeriodicScheduling(period=500)
   public static void measureSpeed(
      @Out("speed") Double speed)
         {…}
}

@Ensemble
@PeriodicScheduling(period=1000)
public class CapacityExchangeEnsemble {

   @Membership
   public static boolean membership(
      @In("coord.position") Coord cPos,
      @In("member.position") Coord mPos)
         {…}

   @KnowledgeExchange
   public static void exchange(
      @InOut("coord.linksCapacities") Map<…> cLinksCapacities,
      @InOut("member.linksCapacities") Map<…> mLinksCapacities)
         {…}
}
```

Figure 3: Code snippets from DEECo component and ensemble specification in Java.

While (i) and (ii) can be obtained by a component directly, the belief about the knowledge of other components (iii) comes as the result of ensemble evaluation and thus is subject to network latencies and limited network connectivity. This contributes to the realistic simulation of the adaptive behavior in decentralized smart CPS.

The switching of the strategies is facilitated by DEECo's "models@runtime" [7] approach, which makes it possible to inspect the architecture of an application at each time instant and modify the architecture as the result of an adaptation strategy.

The runtime model further provides a global view on a DEECo application (including knowledge of components and grouping of components in ensembles). This makes it possible to easily evaluate adaptation strategies by comparing (i) the adaptation taken by a component based on its incomplete (and potentially outdated) belief of the system and (ii) the ideal adaptation that should have been taken if the complete knowledge of the complete up-to-date state of the system and its environment were available.

C. Component and Ensemble Development

DEECo provides a mapping of its concepts (as exemplified by the DSL in Figure 2) to the Java programming language. Figure 3 gives an example of implementing the smart parking scenario in Java. All metadata are captured by annotations, which removes the necessity of having any accompanying specification (in DSL or XML) additional to the Java implementation of components and ensembles.

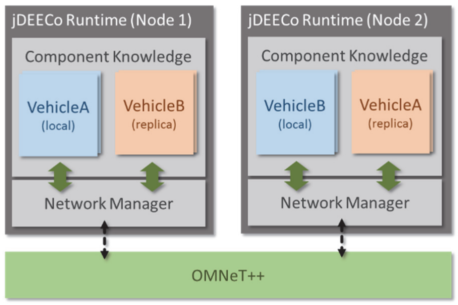


Figure 4: JDEECo runtime.

In the mapping, each component becomes a single class. Its knowledge becomes the class fields and its processes become the static methods. The knowledge that is consumed and produced by a process is specified as process parameters. This processes-knowledge separation allows DEECo runtime to manage snapshotting and atomic updating of knowledge. Similarly, each ensemble is represented as a Java class with a method for membership and a method for knowledge exchange.

IV. JDEECo Simulation Framework

DEECo component model comes with two runtime frameworks – one in Java[1], and one in C++[2]. While the C++ implementation targets actual deployment on embedded devices (e.g., STM32F4 MCU), the Java implementation (which constitutes the artifact presented in this paper) serves primarily for experimentations with autonomous components and self-adaptation. The Java implementation (*JDEECo*) provides a simulation framework which allows experimentations with decentralized adaptive behavior of smart CPS.

The simulation framework is integrated with OMNeT++[3] network simulator. All knowledge exchange passed between components is routed through OMNeT, which provides realistic estimates of network latency w.r.t. to network topology, geographical position of components, network collisions and packet drops, etc. By employing the INET and MIXIM extensions of OMNeT, JDEECo allows for simulating deployments in mixed network environment combining IP networks and mobile/vehicle ad-hoc networks (MANETS / VANETS), as found in modern smart-* systems. Figure 4 illustrates the JDEECo runtime and OMNeT integration.

A simulation of a DEECo application typically requires simulation of responses of the environment (e.g., simulation of car movement in a city). In rapid prototyping, this can be realized at the architectural level by including an "Environment" component, which, by means of models@runtime manipulation, gathers actuation from all components and feeds them with sensing. For more systematic simulations, JDEECo offers a generic sensor/actuator interface and access to the simulation

[1] JDEECo: http://github.com/d3scomp/JDEECo
[2] CDEECo: http://github.com/d3scomp/CDEECo
[3] OMNeT++ : http://www.omnetpp.org/

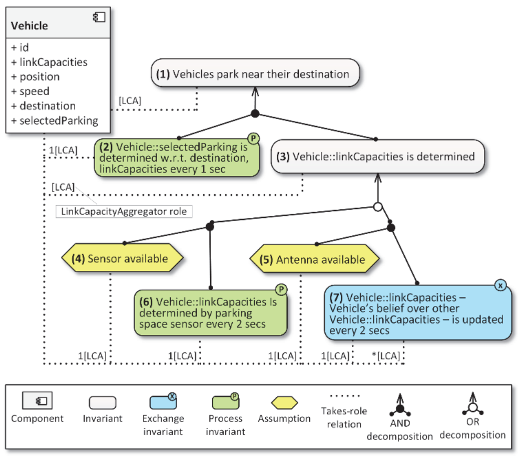

Vehicle
+ id
+ linkCapacities
+ position
+ speed
+ destination
+ selectedParking
(1) Vehicles park near their destination
[LCA]
1[LCA]
(2) Vehicle::selectedParking is determined w.r.t. destination, linkCapacities every 1 sec
(3) Vehicle::linkCapacities is determined
[LCA]
LinkCapacityAggregator role
(4) Sensor available
(5) Antenna available
(6) Vehicle::linkCapacities Is determined by parking space sensor every 2 secs
(7) Vehicle::linkCapacities – Vehicle's belief over other Vehicle::linkCapacities – is updated every 2 secs
1[LCA]
1[LCA]
1[LCA]
1[LCA]
*[LCA]
Component
Invariant
Exchange invariant
Process invariant
Assumption
Takes-role relation
AND decomposition
OR decomposition

Figure 5: IRM tree for the smart parking scenario.

scheduler, which allows plugging-in existing simulators. To date, we have integrated MATSim traffic simulator this way.

V. Invariant Refinement Method

In order to reason about self-adaptation during the design phase, DEECo framework provides the Invariant Refinement Method (*IRM*). IRM is based on goal-oriented requirements elaboration that stems from methodologies such as KAOS [8] and Tropos/i* [9]. IRM captures goals and requirements of the system as *invariants* that describe the desired state of the system-to-be at every time instant. This corresponds to the operational normalcy of the system-to-be and thus aligns well with the need of continuous operation of autonomic component ensembles. IRM is based on iterative decomposition of abstract goals. It is primarily a top-down method, where top-level invariants constitute high-level (general) goals of the application and are further decomposed into more specialized (fine-grained) invariants, which eventually map into concrete component processes (reflecting component responsibilities) and ensembles.

To induce self-adaptivity in architecture design so that the system would react to changing situations in the environment at runtime, IRM captures and exploits architecture variability (in certain potentially overlapping situations) by OR-decompositions. In particular, the designed architecture configurations corresponding to distinct situations that can be encountered at runtime are further elaborated to produce alternative realizations of system requirements.

In Figure 5, the IRM decomposition tree for the example scenario is depicted. There, a simple example of self-adaptation in practice is given. To determine available parking space a Vehicle has two possibilities: (i) to use its own parking space sensor, which is however constrained and provides only readings in the immediate proximity to the vehicle (i.e., current road link), or (ii) to use the information exchanged with other vehicles. As shown in the figure, in order to take advantage of those methods certain assumptions need to hold at runtime – i.e., the reading sensor or the antenna need to remain operational and available for method (i) or (ii), respectively. In the best case, both assumptions hold and as such both methods are used at the same time. However, in cases where there is a problem with ensuring either of those assumptions, the component remains operational and exploits only one of the possible alternatives. By this, Vehicle adapts itself to the situation in the deployment environment and tries to achieve the best possible output, selecting among the available parking spaces near to its destination, always according to the available information.

VI. Summary

To correctly design complex self-adaptive smart CPS is a hard task stemming from the fact that a correct design of such systems has to apply a holistic view that takes into account multiple aspects, many times even conflicting ones.

In this paper, we have briefly introduced DEECo framework, which is intended for development and simulations of such complex self-adaptive smart CPS. In contrast to other frameworks, DEECo (i) is open and easily extensible, (ii) offers a dynamic component model based on ensembles, (iii) has two implementations for experimenting with smart CPS and self-adaptivity, (iv) provides a goal-based design method taking into account self-adaptation, and (v) allows for simulations of real-life deployment by evaluating the system behavior under different network configurations and settings (taking into account also network latency and limited connectivity).

The paper comes together with the artifact containing the JDEECo implementation of the example from Section II and integrated with the JDEECoSim tool. It can be accessed from http://self-adaptive.org/exemplars/v2v-DEECo.

References

[1] *Cyber-Physical Systems: Driving Force for Innovation in Mobility, Health, Energy and Production*. Munich, Germany: National Academy of Science and Engineering, 2011.

[2] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil, "DEECo: An ensemble-based component system," in *Proceedings of CBSE '13*, Vancouver, Canada, 2013, pp. 81–90.

[3] ASCENS, "Autonomic Service-Component Ensembles D4.2: Second Report on WP4," 2012.

[4] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese, "A Language-Based Approach to Autonomic Computing," in *Formal Methods for Components and Objects*, vol. 7542, B. Beckert, F. Damiani, F. de Boer, and M. Bonsangue, Eds. Springer Berlin Heidelberg, 2013, pp. 25–48.

[5] R. De Nicola, A. Lluch Lafuente, M. Loreti, A. Morichetta, R. Pugliese, V. Senni, and F. Tiezzi, "Programming and Verifying Component Ensembles," in *From Programs to Systems. The Systems perspective in Computing*, vol. 8415, S. Bensalem, Y. Lakhneck, and A. Legay, Eds. Springer Berlin Heidelberg, 2014, pp. 69–83.

[6] J. Kephart and D. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[7] G. Blair, N. Bencomo, and R. B. France, "Models@ run.time," *Computer*, vol. 42, no. 10, pp. 22–27, Oct. 2009.

[8] A. van Lamsweerde, "Requirements Engineering: From Craft to Discipline," in *Proceedings of FSE '08*, Atlanta, Georgia, USA, 2008, pp. 238–249.

[9] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos, "Tropos: An Agent-Oriented Software Development Methodology," *Auton. Agents Multi-Agent Syst.*, vol. 8, no. 3, pp. 203–236, May 2004.

4.7 Component-Based Design of Cyber-Physical Applications with Safety-Critical Requirements

Alejandro Masrur,
Michał Kit,
Vladimír Matěna,
Tomáš Bureš,
Wolfram Hardt

In the **Microprocessors and Microsystems** journal.

Accepted manuscript.

The original version is available electronically from the publisher's site at http://dx.doi.org/10.1016/j.micpro.2016.01.007.

Summary of the Paper

The following work presents a method for early-stage time requirements analyzes, ensuring that the developed system satisfies critical properties given in its specification. The analysis proposed are tailored for the DEECo component model and the characteristics of the jDEECo platform - i.e. its execution and communication models (all described in Section 4 of the paper). However, the whole method is designed to be applicable into any development loop of an SCPS. The analysis can be used to suggest a correct system configuration with respect to its time-related aspects. It has been evaluated on the Intelligent Crossing System (ICS) use-case scenario (Section 2), where vehicles are driven automatically by the ICS through the intersection, improving its overall throughput. The idea of the scenario is that the ICS takes control over the approaching vehicles, which then operate in an automated mode unless abnormal situation takes place (e.g. communication fails), which results in returning control to a driver. After the use-case scenario description, the state of the art is presented and the main pitfalls (being addressed by the proposed method) of related approaches are identified. Next (i.e. Section 5), a brief overview on system constraints with respect to its real-time aspects is given. Following, Section 6 describes the actual contribution of the paper where the details of the analysis are given. The goal of this analysis is to provide such a configuration (with respect to its timely aspects - i.e. component process period) of the DEECo-based system that would satisfy its safety-critical requirements. The described analysis accounts for different physical properties of the deployment settings (e.g. maximum velocity of a vehicle) and assumes data prioritization that extends the scenario to allow for instance for privileged vehicles to be recognized in the intersection. The analysis is extended by a brief argumentation on the method robustness considering communication unreliability (i.e. Section 7).

The following part (i.e. Section 8) is the evaluation of the method done in the OMNet++ simulation framework. The scenario is modeled from the perspective of the data traffic in the network and different configurations are implemented in order to verify the validity of the method. Furthermore, in the evaluation part, the measurements taken during the simulation are described by different statistics and depicted in both graphical and tabular forms justifying the validity of the analysis. The paper is then closed by concluding remarks.

Author Contribution and Goals Addressed

This work is a result of a cooperation between the author's host institution and researchers from the Department of Computer Science of the Technical University of Chemnitz. The author's contribution in this work was provisioning the evaluation part for the proposed method. As already mentioned, the evaluation has been based on the OMNet++ simulation in which the ICS scenario has been modeled. The measurements taken during the simulation execution showed the correctness of the proposed analyzes.

With reference to the goals of this thesis, the research effort contributed to this work conforms mostly to goal G3, as it has been the one of the initial attempts to use simulation-based techniques to validate the correctness of a DEECo-based system. Eventually, it has paved the way for the jDEECoSim framework implementation. Moreover, presented analyzes can be seen as a step forward towards formal methods introduction into the DEECo-based system verification.

Component-Based Design of Cyber-Physical Applications with Safety-Critical Requirements

Alejandro Masrur[a], Michał Kit[b], Vladimír Matěna[b], Tomáš Bureš[b], Wolfram Hardt[a]

[a]Department of Computer Science, TU Chemnitz, Germany
[b]Faculty of Mathematics and Physics, Charles University, Czech Republic

Abstract

Cyber-physical systems typically involve large numbers of mobile autonomous devices that closely interact with each other and their environment. Standard design and development techniques often fail to effectively manage the complexity and dynamics of such systems. As a result, there is a strong need for new programming models and abstractions. Towards this, component-based design methods are a promising solution. However, existing such approaches either do not accurately model transitory interactions between components – which are typical of cyber-physical systems – or do not provide guarantees for real-time behavior which is essential in safety-critical applications. To overcome this problem, we present a component-based design technique based on DEECo (Dependable Emergent Ensembles of Components). The DEECo framework allows modeling large-scale dynamic systems by a set of interacting components and, in contrast to approaches from the literature, it provides mechanisms to describe transitory interactions between them. To allow reasoning about timing behavior at the component-description level, we characterize DEECo's closed-loop delay in the worst case, i.e., the maximum time needed to react to a change in the environment. Based on this, we incorporate real-time analysis into DEECo's design flow. This further allows us to analyze the system's robustness under unreliable communication and to design decentralized safety-preserving mechanisms. To illustrate the simplicity and usefulness of our approach, we present a case study consisting of an intelligent crossroad system.

Keywords: Cyber-physical systems, component-based design, safety-critical applications, real-time and timing analysis, unreliable communication, reliability-aware design

1. Introduction

Cyber-physical systems (CPS) are characterized by close interactions with their environment and can be found in different upcoming domains such as smart traffic and transportation, intelligent buildings, smart grid, etc. CPS are inherently distributed, i.e., they rely on a large number of autonomous, typically mobile, embedded devices that form an ecosystem. The joint operation of devices within this ecosystem provides functionality, which is otherwise unattainable by individual devices in isolation.

On the other hand, CPS are highly adaptive, i.e., they constantly react to changes in their environment by modifying its structure and/or behavior accordingly. The collaborative aspect between parts or components of such systems, as well as the necessity for them to function autonomously (in case that, for example, the connection to other parts or components gets interrupted), poses an entire new dimension of challenges for designers. Typically, these challenges are regarded as separate problems of communication networks, distributed control, etc. As such, they have been addressed separately in the respective research fields. However, a significant aspect of modern CPS is that they often are *software intensive* [1], which still remains widely overlooked in the literature.

This means that most of their functionality is embodied in the software, which in turn becomes the most complex and critical constituent. Due to the fact of being distributed and adaptive, software becomes even more complicated and the system starts exhibiting so-called *emergent behavior*. This is the situation where the system's behavior cannot be inferred any longer from its individual parts or components, but their interplay and their joint influence on the environment have to be taken into account.

As a result, there is a strong need for *holistic* design and development methods that rather focus on the whole ecosystem and its overall behavior than on individual constituents. Especially, these methods have to provide systematic software engineering practices that allow managing the increasing complexity of such systems and help controlling emergent behavior. By systematic software engineering, we envision the following four aspects:

i) high-level (requirements-oriented) design with special focus on autonomous behavior, adaptivity and distributed collaboration,

ii) architectural design where a system is modeled by distributed components with clear responsibilities and well-defined interaction patterns,

iii) framework for implementation of components that allows for a straightforward traceability w.r.t. (i) and (ii), and

iv) methods for design-time and runtime analysis (e.g., functional verification, timing analysis) that predict and control the adaptivity and related emergent behavior of these systems.

The basic prerequisite for such systematic software engineering is the existence of software models providing an appropriate level of abstraction. In this respect, classical existing software models (e.g., component-based models such as AUTOSAR [2] or formal process models such as Petri Nets [3]) largely fail to address the needs of distributed adaptive systems. This is mostly because of the fact that they rely on a static structure of the system and thus are unable to model an *open-ended* system, which adapts its architecture to the current state of its environment.

On the other hand, new software models (such as DEECo [4], GCM [5]) appear gradually. They have been specifically developed to capture the nature of distributed adaptive systems and are more suitable for the design and development of CPS. However, by focusing on the cooperative aspects and dynamics of components, they *operate* at a high level of abstraction and do not provide means to model real-time behavior – which is particularly relevant to safety-critical applications.

Contributions. In this paper, continuing and extending our previous work [6], we bridge the gap between a (high-level) component-based description of the system and the analysis of its real-time behavior. In particular, we make use of DEECo (Dependable Emergent Ensembles of Components) in the context of a safety-critical cyber-physical application, viz., an intelligent crossroad system (ICS).

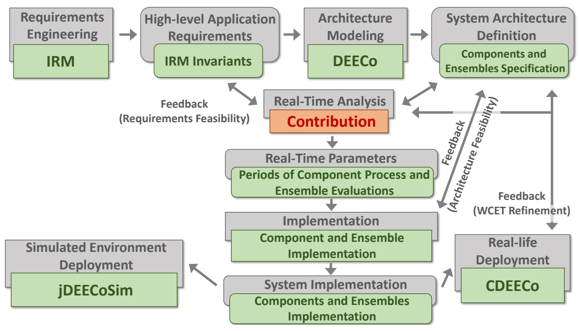

Requirements Engineering
IRM
High-level Application Requirements
IRM Invariants
Architecture Modeling
DEECo
System Architecture Definition
Components and Ensembles Specification
Feedback (Requirements Feasibility)
Real-Time Analysis
Contribution
Feedback (Architecture Feasibility)
Real-Time Parameters
Periods of Component Process and Ensemble Evaluations
Feedback (WCET Refinement)
Implementation
Component and Ensemble Implementation
Simulated Environment Deployment
jDEECoSim
System Implementation
Components and Ensembles Implementation
Real-life Deployment
CDEECo

Figure 1: DEECo's design flow. A sharp rectangle represents a single step of the process. A rounded rectangle depicts outcome of a previous step. While, gray shapes correspond to the general design flow, the green ones are their concrete reifications into the DEECo framework.

DEECo allows for a component-based design of highly dynamic systems and provides deterministic semantics supporting real-time behavior. DEECo is a comprehensive solution as shown in Figure 1, which covers i) requirements engineering with IRM (Invariant Refinement Method) [7], ii) design and development based on DEECo constructs (i.e., components, ensembles, etc.) [4], iii) implementation and deployment with CDEECo (a C++ runtime environment) [8], and iv) simulation and test with the jDEECoSim simulator [9].

We characterize DEECo's closed-loop delay in the worst case, i.e., the maximum delay that it takes to react to a change in the environment. This builds the foundation for our real-time analysis of DEECo-based systems, which we illustrate in the context of our ICS. The results of this analysis are fed back to the component-description level which then capture the application's timing requirements. To this end, we extend DEECo's design flow as shown in Figure 1, where the proposed real-time analysis connects the high-level description with a concrete implementation of the system.

In contrast to [6], in this paper, we provide a more detailed treatment of DEECo's worst-case closed-loop delay and of the ICS case study. In addition, we analyze the system's robustness under unreliable communication and determine an upper bound on the number of packets that may be lost at the communication channel without compromising safety in our case study. Further, we discuss how to implement decentralized safety-preserving mechanisms, which are triggered when something goes wrong, e.g., communication is completely lost, etc. Finally, we validate our analysis by an extensive OMNeT++ simulation considering a varying number of packet losses at the communication channel.

The paper is structured as follows. Section 2 presents our case-study used as the running example, whereas Section 3 discusses related work. In Section 4, we give an overview of DEECo's basic concepts and illustrate them on our case study. Section 5 deals with DEECo's closed-loop delay in the worst case. This is then used to set up our real-time analysis as proposed in Section 6. In Section 7, we study the case of packet losses at the communication channel and discuss how to implement decentralized safety-preserving mechanisms for open-ended CPS. Following, in Section 8, we evaluate our approach by comparing it with results obtained by an OMNet++ simulation and, in Section 9, we present some conclusions.

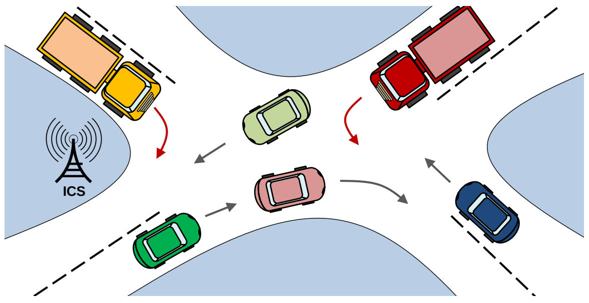


Figure 2: Intelligent crossroad system (ICS)

2. Case-Study

We consider an application scenario in the context of Vehicular Ad-hoc Networks (VANets) [10] and autonomous vehicles, where an ICS optimizes the *car throughput* at a road crossing. This is illustrated in Figure 2, where cars approach a two-lane crossing managed by the ICS.

The idea is to replace traffic lights to a great extent by using car-to-infrastructure (C2I) communication and synchronizing in what order cars cross the intersection for an uninterrupted flow in all directions. Note that there are different ways of implementing this case study. In particular, one can design the ICS to take full control over cars adjusting their speed and steering as considered in [6]. However, this concentrates almost all computation workload at the ICS – making more expensive hardware necessary – and requires cars to be enabled for remote operation.

In this paper, we follow an alternative approach, where the ICS computes – taking traffic conditions and regulations into account – and *assigns* a speed to an approaching car at the intersection. The car will then have to keep this speed to cross the intersection without stopping. This solution is more viable to implement the proposed case study, since it does not require cars to be modified for a remote operation. On the contrary, each (autonomous) car is responsible for driving along its own trajectory and keeping its speed as assigned to it. If necessary, however, a car may stop to avoid a crash or accident.

The ICS assigns speeds to cars and these report their current speeds back to the ICS via C2I communication. As discussed later, the ICS needs to keep track of cars' speeds in order to detect potential hazards and trigger safety mechanisms. To this end, we foresee two operation modes:

i) Automatic mode: This is the default mode where all cars at the crossing behave as expected, i.e., they keep their corresponding speeds as computed by the ICS. Here, the highest possible *throughput* is reached provided that speed limit regulations are observed and safety of all traffic participants can be guaranteed.

ii) Manual mode: This is the exception mode where one or more cars do not keep their computed speeds and/or there is no communication between a car and the ICS, e.g., conventional cars with a human driver. In this mode, the ICS works as standard traffic lights.

We define the *region of influence* by the area in which the ICS *controls/monitors* all approaching cars. We assume that this area consists of a 50 m radius around the intersection. In this region, vehicles are prioritized such that their priorities increase as they get closer to the center of the intersection and drop when they move away of it. Some of the vehicles at the ICS might also be *privileged* such as, for example, ambulances or police cars in an emergency situation, etc.

The ICS can detect when a car enters the region of influence, e.g., by radar, pressure sensors, etc. If no communication is received from one car after it has entered the region of influence (in particular, a car's intended direction, its current speed, etc. are needed), the ICS assumes that either there has been an error or it is a conventional car with no I2C communication and switches to manual mode. The same happens if communication is lost to one or more cars; a more detailed analysis of this is given in Section 7.

Pedestrians can be easily handled in the manual mode. In the automatic mode, the ICS can detect when pedestrians stand at the crossing for some time, e.g., by pressure sensors, request buttons, etc., and stop the traffic to let them cross in a safe manner. Again, each car is responsible for itself and should be able to react to unpredicted situations, e.g., performing an emergency break, according to valid traffic regulations.

This scenario exhibits different challenges that need to be faced when designing dynamic distributed systems. One of those challenges is the description of architectural changes that occur during runtime. In our case study, cars/vehicles arrive to and leave the system at different points in time, their priorities vary according to their distances to the crossing, etc. Such details need to be properly reflected in the system design.

Furthermore, this scenario exhibits real-time requirements imposed to the system. In particular, it is required that the *reaction time* between a car and the ICS is kept below a certain upper bound in order to ensure the required responsiveness of the overall system, where unreliable communication needs to be considered. In turn, meeting those real-time requirements allows us to guarantee safety, which translates into a collision-free crossing in our case study.

Lastly, since it is not possible to guarantee a fully reliable communication, the system has to be designed to be self-adaptive. This way, the system switches to manual mode when it realizes that real-time requirements cannot be met. To this end, as discussed later, we configure a *watchdog timer* at all components (cars and ICS) that triggers a switch to manual mode.

3. Related Work

To position the presented approach among a multitude of the existing component models, we constrain our focus to those that enable analyzing timing aspects.

The most prominent example is certainly AUTOSAR [2], which is of common use in the automotive industry. AUTOSAR serves as a specification for different layers (i.e., application software, runtime environment and basic software) of a system constituted by hierarchical components. AUTOSAR itself does not provide any means to perform timing analysis and for that reason it has been enriched by the TIMing MOdel (TIMMO) [11], which builds on the Timing Augmented Description Language (TADL).

Another widely used model supporting timing analysis is AADL (Architecture Analysis and Design language) [12]. It relies on Real-Time Calculus (RTC) [13], which is a formalism that allows for system-level performance analysis of stream-processing systems constrained by hard real-time requirements. Essentially RTC models are extracted from AADL and subsequently the RTC tools can be employed.

Similarly, timing analysis enabled at the model level are supported by the BIP (Behavior, Interaction, Priority) framework [14]. BIP supports real-time aspects by using *timed components*, which allow for timing properties being specified using *timed variables* and *transitions*. Those are accounted for during the validation within the real-time engine implementing operational semantics of the BIP models.

An architectural approach to modeling systems is also taken by SysML, which integrates with MARTE [15] to enable modeling non-functional properties such as power consumption, performance and timing.

AUTOSAR, AADL, SysML and BIP assume, however, static component architectures, which effectively hinders their application in case of modern CPS development. In contrast, our approach targets at open-ended CPS where the architecture changes continuously (e.g., cars appear and disappear without anticipation) leading to emergent behavior in the system.

Another example of a component model that allows for timing estimates of a system is the Palladio Component Model (PCM) [16]. The strongest point of PCM is the extra-functional property prediction framework that allows estimating overall system performance. It relies on different models, depending on what is required to be analyzed (e.g., reliability, performance, throughput, etc.). These are decorated by non-functional properties specification, which serves as an input for model analysis. Similar to our approach PCM builds on simulation-based techniques for model validation, which allows exploring different designs for a given system or application. However, in a similar manner as the related approaches above, PCM does not support dynamic architectures, which again limits its applicability in today's complex CPS.

An interesting approach is proposed by Etzien et al. in [17, 18]. The authors describe a modeling method for evolutionary distributed systems using the concept of System of Systems (SoS). In order to capture both static and dynamic properties of the developed system, they use the contract paradigm for specifying legal configurations of a SoS and to describe architectural changes during runtime. In [19], the authors extend their work by providing a method for schedulability analysis. They based their technique on both analytical and model checking methods, which combined with the SoS contracts provide for a compositional and scalable solution.

In order to perform the analysis of [19], one needs to deal with a full-fledged implementation of the system, from which necessary parameters are extracted to construct a state machine for analysis. Such method is rather suitable for verification and validation of an existing implementation. In our case, the proposed analysis is intended to be used at an early design phase where mostly system requirements are known (see Figure 1). Moreover, our analysis addresses CPS with a strong connection to the environment – see Section 6, whereas [19] focuses on more general-purpose systems.

4. Modeling with DEECo

As stated above, we make use of the DEECo component model [4]. DEECo describes the architecture of a CPS by means of components (i.e., encapsulated well-defined active entities, which perform sensing, computation and actuation) and so called ensembles, which are dynamically established groups of components that cooperate to achieve a particular goal. DEECo further provides a special requirements engineering method and traceability of requirements to components and ensembles – for further details, we refer to [7].

4.1. Architecture Modeling

Components. To illustrate the principles behind DEECo, Listing 1 depicts a component using a DSL (Domain-Specific Language) description.[1] In DEECo, each component consists of *knowledge* – see lines 8-16 – reflecting its current state. Knowledge is expressed by attributes organized into hierarchical data structures. Access to one or more such attributes of a component is performed through interfaces – see lines 1-5 – that are featured by the component.

```
1  interface MovingUnit:
2      id, time, crossingId, crossingDistance, crossingDirection, speed, privileged, mode
3
4  interface MovingUnitAggregator:
5      id, time, vehicles, speeds, mode
6
7  component Vehicle features MovingUnit
8      knowledge:
9          id: 42,
10         time: 1440691842456 ms,
11         crossingId: 12
12         crossingDistance: 35 m,
13         crossingDirection: South−West,
14         speed: 50 Km/h,
15         privileged: FALSE,
16         mode: AUTOMATIC,
17         ...
18     process UpdateSpeed:
19             in speed
20         function:
21             Actuators.setSpeed(speed);
22         scheduling: periodic( 5 ms )
23     ...
24
25 component ICS features MovingUnitAggregator
26     knowledge:
27         id: 12,
28         time: 1440691842458 ms,
29         vehicles: [...],
30         speeds [...]
31         privileged: [...],
32         mode: AUTOMATIC,
33         ...
34     process findPrivilegedVehicles:
35             in vehicles, inout privileged
36         function:
37             for (v : vehicles)
38                 if (v.privileged)
39                     privileged.add(v)
40         scheduling: triggered( changed(movingUnits) )
41     ...
```

Listing 1: DEECo component definitions based on a DSL

In addition, each component has a set of processes (essentially real-time tasks) that manipulate its knowledge. A process is characterized by a function (e.g., lines 20-22), whose parameter list consists of knowledge attributes. DEECo's *runtime environment* manages the release of processes and takes care of knowledge retrieval before a process is executed and knowledge update – also referred to as knowledge exchange – when a process finishes executing. Each of the component's processes is executed in isolation meaning that it is not supposed to communicate with other processes (either from the same or a different component) in any other way than via the component's knowledge.

[1]Note that this DSL specification serves for demonstration only. Later we discuss how to derive a C++ implementation from this specification, which can then be used on embedded devices.

A process can be executed in response to a timer event (i.e., periodic execution) or as a reaction to a change in one of its attributes. In our example, the *Vehicle* component has a process that sets/updates the speed of the car or vehicle. This is repeated periodically every 5 ms (see line 22). As another example the ICS process, shown in Listing 1, determines whether there are privileged vehicles in its region of influence (lines 34-40) and is executed whenever the number of vehicles changes (line 40). Once processes are released (by DEECo's runtime environment), these are handed over to the platform's operating system (OS), which is responsible for scheduling them according to a desired policy – see Section 4.2.

Ensembles. An ensemble in DEECo defines a semantic connector between components and constitutes their *composition*. The composition in DEECo is *flat* and occurs implicitly by components dynamically joining an ensemble at runtime. When specifying an ensemble, prospective components are described by roles. One component in the ensemble has a *coordinator's* role, whereas the remaining components are *members* of the ensemble.

The roles are defined by the interfaces – in our example, *MovingUnit* and *MovingUnitAggregator* – which are matched at runtime to the actual components (i.e., their knowledge) for a structural coincidence. Later, those components with matching interfaces are considered for the ensemble evaluation process, which is composed of two steps. The first step involves checking the *membership condition*, which is expressed as a logic predicate formulated upon coordinator's and member's attributes. This determines whether two components (a coordinator and a member component) should form an ensemble. The second step depends on the results of the membership condition check and consists of exchanging attribute values between coordinator and member according to the description given in the *knowledge exchange* specification.

In the example in Listing 2, the coordinator role is determined by the interface definition *MovingUnitAggregator* and the member role by *MovingUnit*. This way, during the ensemble evaluation, only components featuring appropriate interfaces will be considered. The membership condition further constraints the number of ensemble members to those, which are located no more than 50 m from the coordinator's location. Then, according to the knowledge exchange description, the coordinator's *movingUnits* attribute is updated with information about all components that fulfill the membership condition (which is checked every $p_{ens,i}$ time units – see line 8 – with *i* being an index representing the component). This way, the ICS is aware only of those vehicles, which are currently in its close proximity.

```
1  ensemble UpdateMovingUnitInformation:
2      coordinator: MovingUnitAggregator
3      member: MovingUnit
4      membership:
5          coordinator.id = member.crossingId, member.crossingDistance < 50 m
6      knowledge exchange:
7          coordinator.movingUnits.add({member})
8      scheduling: periodic( p_ens,i )
```

Listing 2: A DSL example of an ensemble definition.

DEECo's deterministic semantics. DEECo components are autonomous and rely only on data that is present in their knowledge. As mentioned before, any interaction of a component with other components is realized by ensembles, which is *externalized* from the component itself. This property of DEECo's component model suits very well to both the design and the implementation of distributed adaptive systems, since all technical aspects related to communication between remote components are abstracted away from the design phase and left for the runtime environment to deal with them.

Technically, the runtime environment periodically *propagates* ensemble-relevant knowledge to all other components or nodes in the system – note that gossip-based algorithms [20] might be used for this purpose. In our case study, ensemble-relevant data are the car's distance to the crossing, its speed, and its intended direction, etc. This is used to evaluate whether cars are heading in the direction of the crossing or not.

Each node then keeps relevant reference knowledge from all other nodes from which it has received data. In other words, components or nodes join (and leave) the system in an implicit manner without performing any explicit *handshaking*. Since ensemble-relevant information is present at all nodes, DEECo's runtime environment performs a local evaluation of an ensemble membership condition. If this holds true, the local reference knowledge of the remote components involved is used for the *knowledge exchange* process.

In this way, DEECo's semantics separates *decision taking* (i.e., ensemble evaluation and its eventual knowledge exchange) from *information sharing* (i.e., knowledge propagation) processes at the components. Since a DEECo component takes decisions based on locally available data, it does not need to synchronize with other components in the system. On the one hand, this has the advantage of high flexibility and adaptability. On the other hand, clearly, local data might get outdated at the different nodes, which needs to be analyzed carefully as illustrated in Section 5.

```
namespace VehicleComponent {
struct Knowledge: CDEECO::Knowledge {
    VehicleId id;
    Time time;
    CrossingId crossingId;
    Distance crossingDistance;
    Direction crossingDirection;
    Speed speed;
    bool privileged;
    Mode mode;
    ...
};

class UpdateDistance: public
        CDEECO::PeriodicTask<Knowledge, Distance> {
    UpdateDistance(auto &component);
    Distance run(const Knowledge in);
};

class SetSpeed: public CDEECO::PeriodicTask<Knowledge, void> {
    SetSpeed(auto &component);
    void run(const Knowledge in);
};

class Component: public CDEECO::Component<Knowledge> {
    static const CDEECO::Type Type = 0x00000001;

    UpdateDistance updateDistance = updateDistance(*this);
    SetSpeed setSpeed = SetSpeed(*this);
    ...

    Component(CDEECO::Broadcaster &broadcaster,
        const CDEECO::Id id,
        bool remotelyOperable);
};
}
```

Listing 3: A C++ example of Vehicle component.

4.2. Implementation and deployment

The implementation and distributed deployment of DEECo systems are supported by the CDEECo framework. This framework maps DEECo concepts to C++ and constitutes our run-

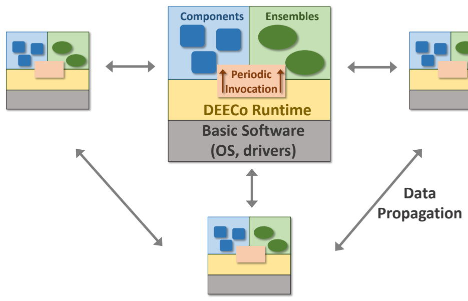


Figure 3: DEECo distributed deployment.

time environment (taking care of periodically propagating knowledge, performing ensemble evaluations, performing knowledge exchange if applicable, etc.). As depicted in Figure 3, CDEECo relies on an OS providing hardware abstraction and other services. Clearly, this OS needs to support real-time behavior, i.e., real-time scheduling, interrupt handling, etc., to be used in safety-critical applications.

With respect to CDEECo's implementation in C++, components and processes are handled as *classes*, while knowledge is treated as a static data structure (with a fixed size in bytes). Thus it is possible to operate on knowledge within bounded time. Moreover, it is easy to fragment knowledge to fit into network packets, also within bounded time.

```
namespace VehicleInfoAndSpeedExchange {
typedef CDEECO::Ensemble< ICS::Knowledge, Vehicle::Knowledge*,
        Vehicle::Knowledge, Speed> EnsembleType;

class Ensemble: EnsembleType {
    Ensemble(CDEECO::Component<ICS::Knowledge> &coord,
        CDEECO::KnowledgeLibrary<Vehicle::Knowledge> &lib);

    Ensemble(CDEECO::Component<Vehicle::Knowledge> &mbr,
        CDEECO::KnowledgeLibrary<ICS::Knowledge> &lib);

     bool isMember(const CDEECO::Id coordId,
        const ICS::Knowledge coordKnowledge,
        const CDEECO::Id memberId,
        const Vehicle::Knowledge memberKnowledge);

    Vehicle::Knowledge* memberToCoordMap(
        const ICS::Knowledge coord,
        const CDEECO::Id memberId,
        const Vehicle::Knowledge memberKnowledge);

    Speed coordToMemberMap(
        const Vehicle::Knowledge member,
        const CDEECO::Id coordId,
        const ICS::Knowledge coordKnowledge);
};
}
```

Listing 4: A C++ example of vehicle-crossroad ensemble

A component is represented by a class which inherits *basic behavior* from a template with *knowledge* as an argument. Similarly, processes are represented by classes inheriting basic be-

10

havior from a template, depending on the process type, while accepting input and output types as template arguments. The process's code is contained in a *virtual method* in the base class. The component class, process classes, and the knowledge data structure can be wrapped in a *namespace* to improve code readability as shown in Listing 3.

Similarly to components, ensembles are also implemented using classes inheriting basic behavior from a template. The membership method and a pair of knowledge mapping methods are realized as virtual methods in the base class. In case of ensembles, template arguments are more complex, since the data type have to capture the coordinator's and the member's input/output *knowledge types*. In order to simplify the code, the complex type of the ensemble can be defined using typedef and wrapped together with ensemble class in a *namespace* as displayed in Listing 4.

CDEECo's sources are located on GitHub[2]. So far, we have been using FreeRTOS[3] as an OS to schedule DEECo processes on the corresponding nodes. As mentioned above, CDEECo periodically propagates knowledge over available communication channels, in our case study, a VANet. To this end, it periodically broadcasts *binary patches* covering the whole corresponding component's knowledge. These are bulks of binary data with offset, size and source component ID (i.e., an identification code). Patches are then used to update knowledge on the receiver component when no packet is lost. Otherwise the knowledge can be just partially updated.

This solution was chosen to maximize knowledge propagation, while the consistency can be achieved by storing dependent data in single packets. Knowledge that has been successfully received from remote components is stored in a so-called *knowledge library*. A knowledge library is a data structure holding predefined number of remote knowledge data sets; when an ensemble is evaluated true, a knowledge exchange is triggered copying (locally available) data from the knowledge library to the local component's memory space.

5. DEECo's Closed-Loop Reaction Time

In this section, we analyze DEECo's closed-loop reaction time in the worst case. This is defined as the maximum delay that it takes a DEECo-based system to react to changes in the environment. The term *closed-loop* reflects the fact that DEECo components interact with one another.

For example, if a component A experiences a change in its internal states, e.g., due to one or more physical variables measured by its sensors, this will take some time to reach another component B – connected by ensembles – in the system. Similarly, component B's reaction to the change in component A will take some additional time to reach back component A. The sum of these two times is the closed-loop delay between A to B. In other words, component A and B form a loop.

For ease of exposition, we first make use of our case study and then generalize our results to make them independent of the application. In our case study, knowledge needs to be exchanged from a car to the ICS and from the ICS back to the car for the system to work as expected. However, knowledge exchange happens based on *local data* when the corresponding ensemble condition is evaluated to true at both the ICS and the car nodes separately.

As discussed above, knowledge is propagated (from the car to the ICS and vice versa) and the ensemble membership check is performed (at the car and at the ICS) on a periodic basis. Let

[2]`https://github.com/d3scomp/CDEECo`
[3]`http://www.freertos.org/`

11

us denote by $\hat{p}_{pro}$ the maximum period with which knowledge is propagated by any node in the system. Similarly, let $\hat{p}_{ens}$ be the maximum period with which ensembles are evaluated at any node in the system. That is:

$$\hat{p}_{pro} = \max_{\forall i}\left(p_{pro,i}\right),$$
$$\hat{p}_{ens} = \max_{\forall i}\left(p_{ens,i}\right),$$

where $p_{pro,i}$ and $p_{ens,i}$ are the knowledge propagation and the ensemble evaluation period of a node i, respectively, with i being an index that identifies the corresponding component/node.

DEECo eliminates the need for synchronization and handshaking between components while enormously simplifying the design and development complexity. As a result, these two processes are not synchronized with one another and we have the following conditions in the worst case:

i) A car propagates its knowledge to the ICS immediately after a membership evaluation has been performed at the ICS. As a result, data is received $\hat{p}_{ens}$ time later at the ICS, when the next membership evaluation is performed.

ii) In a similar manner, the ICS propagates its knowledge to the corresponding car just after a membership evaluation has been performed at the car. As a result, data is received $\hat{p}_{ens}$ time later at the car too.

iii) The knowledge of the car changes immediately after knowledge has been propagated to the ICS. As a result, the *current* knowledge is propagated with a delay $\hat{p}_{pro}$ from the car to the ICS, when a new propagation is performed.

iv) The ICS's knowledge changes immediately after knowledge has been propagated to the car. Hence, the *current* knowledge is not propagated until a new propagation is started $\hat{p}_{pro}$ time later.

As a result, in the worst case, we have a delay due to the asynchronous nature of the DEECo framework which is given by the following expression:

$$2 \times \hat{p}_{pro} + 2 \times \hat{p}_{ens}. \tag{1}$$

In addition, there is also a process running at the ICS which computes the speed for the car that guarantees no collisions at the current traffic situation. This process is triggered when a knowledge exchange is executed at the ICS (i.e., when the ensemble is evaluated to true between the car and the ICS). We denote by r_{ICS} the worst-case response time (WCRT) of this process. Analogously, there is a process running at the car, which applies the new speed values to the *physical* car. This process is also triggered when a knowledge exchange happens at the car and its WCRT is r_{car}.

As a result, the worst-case delay D_{max} for a closed-loop reaction in DEECo, i.e., a reaction of a car to an input from the ICS computed based on the car's current knowledge, is given by the following equation also illustrated in Figure 4:

$$D_{max} = 2 \times \hat{p}_{pro} + 2 \times \hat{p}_{ens} + c_{ICS} + c_{car} + r_{ICS} + r_{car}, \tag{2}$$

where c_{car} is the communication delay from the car to the ICS and c_{ICS} is the communication delay from the ICS to the car.

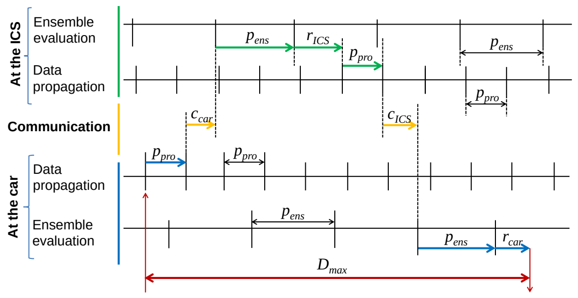


Figure 4: Composition of DEECo's closed-loop delay D_{max}: In the worst case, data may change immediately after knowledge has been propagated at the car. This data may also arrive after an ensemble evaluation has been performed at the ICS. In addition, computation at the ICS may finish immediately after knowledge propagation and the resulting data then reaches the car just after the end of an ensemble evaluation.

Since there is interference by other messages (from other cars), c_{car} is the maximum possible communication delay in the network. However, from the ICS to the car, there is no interference – assuming a full-duplex communication channel – and the communication delay c_{ICS} is equal to the transmission time, since the ICS does not compete for accessing the network.

The term $2 \times \hat{p}_{pro} + 2 \times \hat{p}_{ens}$ in Equation (2) is clearly intrinsic to DEECo and does not depend on the application, but rather on how components are configured. In addition, $c_{ICS} + c_{car}$ is the total communication delay between the ICS and a car, whereas $r_{ICS} + r_{car}$ is the delay due to computation at the ICS and at the car. As a result, DEECo's closed-loop delay in the worst-case can be generalized, i.e., made independent of the application under consideration, as follows:

$$\bar{D}_{max} = 2 \times \hat{p}_{pro} + 2 \times \hat{p}_{ens} + C_{max} + R_{max}, \quad (3)$$

where C_{max} is the sum of the worst-case communication delay between any two DEECo components and R_{max} is the sum of the WCRTs of computation processes involved at the corresponding components or nodes.

6. Real-Time Analysis

The purpose of performing a real-time analysis is to guarantee that timing constraints can be met by the system, which is required in safety-critical applications. In the case of DEECo, this boils down to checking that $\bar{D}_{max}$ as per Equation (3) is below a given time upper bound, which stems from physical processes involved and needs to be known to or obtained by the designer. In turn, $\bar{D}_{max}$ depends on C_{max} and R_{max}, i.e., on the delays incurred by communication and computation processes involved – which are closely related to the technologies and techniques used for implementing the system.

All in all, our real-time analysis consists of the following steps that we illustrate next in the context of our case study: i) obtaining the worst-case computation delay, ii) obtaining the worst-case communication delay, iii) determining system constraints, and iv) obtaining a feasible DEECo configuration.

6.1. Obtaining the worst-case computation delay

As discussed previously, CDEECo – DEECo's runtime environment – is executed on top of an OS at each node in the system. Among others, CDEECo is in charge of releasing component's processes as specified in the component description. Clearly, to be used in safety-critical applications, CDEECo relies on specific technologies that make real-time scheduling and real-time communication possible. In particular, the OS needs to support real-time scheduling; otherwise, it will not be possible to guarantee real-time behavior.

The techniques for schedulability analysis, i.e., testing whether processes meet their deadlines at the different nodes, strongly depend on the scheduling algorithm used. As mentioned above, CDEECo makes use of FreeRTOS, which supports fixed-priority scheduling and allows for the *rate monotonic* policy [21]. That is, processes are given fixed priorities according to the following rule: The shorter a process's period is, the higher the priority assigned to it.

Let us denote by $\mathbf{T}$ the set of processes on a given node. Further, τ_i is a process that belongs to $\mathbf{T}$ where e_i denotes its worst-case execution time (WCET) and p_i denotes its period of repetition. For $\mathbf{T}$ to be schedulable, the following has to hold for each τ_i and $1 \leq i \leq |\mathbf{T}|$, being $|\mathbf{T}|$ is the number of elements in $\mathbf{T}$:

$$\sum_{\forall \tau_j \in \widehat{\mathbf{T}}} \left\lceil \frac{p_i}{p_j} \right\rceil e_j \leq p_i, \tag{4}$$

where by $\widehat{\mathbf{T}}$ we denote the subset of processes from $\mathbf{T}$ which have a higher priority than τ_i.

This expression means that for each process τ_i to be schedulable (and, hence, for $\mathbf{T}$ to be schedulable), the sum of all executions of higher-priority processes in a time interval equal to p_i plus its own execution e_i should be less than its deadline p_i. Note that Equation (4) is sufficient but not necessary. A sufficient and necessary test can be achieved by response time analysis [22]; however, the sufficient test of Equation (4) is enough for the purpose of this paper. Now, since the following holds:

$$\sum_{\forall \tau_j \in \widehat{\mathbf{T}}} \left\lceil \frac{p_i}{p_j} \right\rceil e_j \quad \leq \quad \sum_{\forall \tau_j \in \widehat{\mathbf{T}}} \left(\frac{p_i}{p_j} + 1 \right) e_j,$$

we can reshape Equation (4) to:

$$\sum_{\forall \tau_j \in \widehat{\mathbf{T}}} \frac{e_j}{p_j} + \frac{\sum_{\forall \tau_j \in \widehat{\mathbf{T}}} (e_j)}{p_i} \leq 1. \tag{5}$$

Clearly, if Equation (5) holds, Equation (4) will also hold. However, Equation (5) is easier to compute and operate with.

In our case study, to obtain $R_{max} = r_{ICS} + r_{car}$, let us denote by $\mathbf{T}_{ICS}$ the set of computation processes at the ICS. Further, we assume that a process τ_i is the one involved in the closed-loop delay, i.e., the one computing the new speed for a given car. Considering that $\widehat{\mathbf{T}}_{ICS}$ is the subset with higher- or equal priority processes than τ_i at the ICS, we can compute r_{ICS} as follows:

$$r_{ICS} = \left(\sum_{\forall \tau_j \in \widehat{\mathbf{T}}_{ICS}} \frac{e_j}{p_j} \right) p_i + \sum_{\forall \tau_j \in \widehat{\mathbf{T}}_{ICS}} e_j. \tag{6}$$

14

Analogously, at the car, we can obtain r_{car} as follows:

$$r_{car} = \left(\sum_{\forall \tau_j \in \widehat{\mathbf{T}}_{car}} \frac{e_j}{p_j} \right) p_i + \sum_{\forall \tau_j \in \widehat{\mathbf{T}}_{car}} e_j. \tag{7}$$

6.2. Obtaining the worst-case communication delay

Similar to computation delay, communication delay strongly depends on the underlying technologies and techniques used. In the context of our case study, a number of techniques have been already proposed to realize collision-free communication or reduce packet loss in VANets. For example, space division multiple access (SDMA) has been proposed for busy intersections, which is based on assigning time slots to given locations on a road [23, 24]. That is, cars use different time slots to communicate depending on where on the road they currently are. Other solutions combine special antennas with a TDMA (time division multiple access) scheme to reduce packet loss in a VANet [25, 26].

Since VANets are usually based on wireless communication [27], we assume that Ethernet IEEE 802.1Q is the underlying protocol, which provides mechanisms to prioritize *messages* [28]. In general, we will normally have a number of access points (AP) which are connected to a full-duplex switch via Ethernet. In this section, we consider that communication to the AP is collision-free as per one of the above mentioned approaches – later we remove this assumption to analyze the effect of packet loss. Assuming that wireless network provides 100 Mbps and that messages are at most 1 Kbit (1024 bits), then the transmission time c_W on the wireless network is at most – considering a 144-bit protocol overhead:

$$c_W = \frac{1024 + 144}{100\,\text{Mbps}} = 11.68\,\mu\text{s}.$$

However, the switch then sends messages to the ICS according to their priorities. Let us consider that the ICS's region of influence is divided into sectors with different priorities – see Figure 5. Cars that are in the first sector (e.g., within 10 m from the intersection) have higher priority than cars in the second sector (e.g., from 10 m to 20 m) and so on. At a given point in time, if a car is in more than one sector simultaneously, it will be assigned the highest priority among those sectors. The switch then sends messages to the ICS according to these priorities.

Let us now analyze the communication segment between the switch and the ICS. To this end, let $\mathbf{M}$ denote the set of all messages being sent to the ICS over the switch. Further, m_i denotes one such message in $\mathbf{M}$ where c_i is its transmission time – note that c_i is constant for a given i which results from the amount of bits to be sent and the bandwidth of the communication channel – and z_i denotes the minimum inter-arrival time between two consecutive such messages. The deadline of a message is also given by z_i.

Generally, for all messages in $\mathbf{M}$ to meet their deadlines, the following has to hold for $1 \leq i \leq |\mathbf{M}|$, where $|\mathbf{M}|$ is the number of elements in $\mathbf{M}$:

$$b_i + \sum_{\forall m_j \in \widehat{\mathbf{M}}} \left\lceil \frac{z_i}{z_j} \right\rceil c_j \leq z_i, \tag{8}$$

where $\widehat{\mathbf{M}}$ is the subset of $\mathbf{M}$ with higher- or equal priority messages than m_i and b_i denotes *blocking time* on the communication channel. That is, whenever a message needs to be sent, a

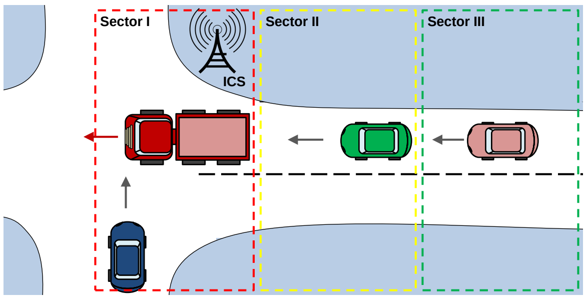


Figure 5: Priorities are given according to the proximity to the intersection

lower-priority message might eventually be using the communication channel. Since this lower-priority message cannot be interrupted, there is a blocking time on the bus. Clearly, in worst case, b_i is given by the maximum transmission time among all lower-priority messages:

$$b_i = \max_{l=i}^{|\mathbf{M}|}(c_l). \tag{9}$$

Considering that $\sum_{\forall m_j \in \widehat{\mathbf{M}}} \left\lceil \frac{z_i}{z_j} \right\rceil \leq \sum_{\forall m_j \in \widehat{\mathbf{M}}} \left(\frac{z_i}{z_j} + 1 \right)$ holds, we can remove the ceiling function and approximate Equation (8) as shown below:

$$\sum_{\forall m_j \in \widehat{\mathbf{M}}} \frac{c_j}{z_j} + \frac{b_i + \sum_{\forall m_j \in \widehat{\mathbf{M}}}(c_j)}{z_i} \leq 1. \tag{10}$$

To demonstrate this in our case study, let us assume that the Ethernet link between the switch and the ICS has a bandwidth of 1 Gbps. If messages have a length of at most 1 Kbit (1024 bits), and the protocol overhead is of 144 bits, we have that the transmission time c_i of a message m_i is given by $\check{c}_E$:

$$\check{c}_E = \frac{1024 + 144}{1\,\text{Gbits}} = 1.168\,\mu\text{s}.$$

Further, with help of Equation (10), we can compute the transmission time on the Ethernet link taking contention by higher- and equal priority messages into account, which we denote by $\hat{c}_E$:

$$\hat{c}_E = \left(\sum_{\forall m_j \in \widehat{\mathbf{M}}} \frac{c_j}{z_j} \right) z_i + b_i + \sum_{\forall m_j \in \widehat{\mathbf{M}}} c_j. \tag{11}$$

In addition to the transmission time, there is always a delay at the AP and at switches in Ethernet – denoted by e_{AP} and e_{SW} respectively, which accounts for buffering and routing tasks. This is typically in the order of $2\,\mu$s.

$C_{max} = c_{ICS} + c_{car}$ can now be obtained. To this end, recall that we have two Ethernet links: one from the AP to the switch and another from the switch to the ICS. The ICS does not suffer from contention at the communication channel, since the connection from and to the APs is assumed to be full duplex. As a result, c_{ICS} is given by:

$$c_{ICS} = 2 \times \check{c}_E + c_W + e_{SW} + e_{AP}. \tag{12}$$

On the other hand, cars share the communication channel and, hence, they may have contention at the communication channel leading to a c_{car} as follows:

$$c_{car} = 2 \times \hat{c}_E + c_W + e_{SW} + e_{AP}. \tag{13}$$

6.3. Determining system constraints

Timing constraints are clearly derived from the application. In our case, we consider that a car needs to be provided with new speed values at every single meter of its trajectory (taking vehicle dynamics into account such inertia, braking distance, etc.). If a car's speed is at maximum 50 Km/h (assuming an urban scenario), we need to compute the time $t_{1\mathrm{m}}$ that it needs to cover 1 m of its trajectory:

$$t_{1\mathrm{m}} = \frac{1\,\mathrm{m} \times 3600\,\mathrm{s/h}}{50 \cdot 10^3\,\mathrm{m/h}} = 72\,\mathrm{ms}. \tag{14}$$

On the other hand, the computation and communication overhead depend on the number of components, in particular, cars/vehicles in the system, which is the second constraint from the application. Clearly, the more cars enter the ICS's region of influence, the more computation and communication overhead there will be. To compute the maximum possible number of cars at the crossing, let us assume that a car is at least 2 m long and that there is a least a 1 m distance between any two cars. As a result of this, in the worst possible case, the number of cars n approaching the intersection from all directions is given by the following equation:

$$n = 4 \times \left\lceil \frac{50\,\mathrm{m}}{3\,\mathrm{m}} \right\rceil = 68. \tag{15}$$

We can use Equation (15) to configure $\hat{p}_{ens}$ and $\hat{p}_{pro}$ in the next section and the timing constraint as per Equation (14) to perform a feasibility analysis as discussed later.

6.4. Obtaining a feasible DEECo configuration

There will be at most 68 different ensemble instances (between the ICS and each of the cars) at the ICS – see Equation (15). In addition, there will be 68 processes to compute new speed values for each car. Since the ensemble membership check triggers a knowledge exchange – recall that knowledge exchange is based on locally available data and that knowledge/data propagation (from the ICS to the cars and vice versa) is a separate and asynchronous process – when evaluated true, we can assume that in the worst case all 68 ensemble processes trigger their corresponding computation processes simultaneously. In addition, there will be one knowledge propagation process for the ICS[4].

Assuming that all processes have a WCET $e_i = 25\mu\,\mathrm{s}$ (note that most these processes consist in checking logic conditions, assigning pointers to given memory spaces, etc., or are simple computations), we can use Equation (5) applied to the ICS as follows:

$$\frac{0.025\,\mathrm{ms}}{\check{p}_{pro}} + 68 \cdot \frac{2 \times 0.025\,\mathrm{ms}}{\check{p}_{ens}} + \frac{(0.025\,\mathrm{ms} + 68 \cdot (2 \times 0.025\,\mathrm{ms}))}{\check{p}_{ens}} \leq 1, \tag{16}$$

[4]Note that the knowledge propagation from cars does not produce any overhead at the ICS, but at the respective cars.

where $\check{p}_{pro}$ and $\check{p}_{ens}$ are the minimum periods with which knowledge is propagated and with which ensembles are evaluated in the system. That is:

$$\check{p}_{pro} = \min_{\forall i} \left(p_{pro,i}\right),$$
$$\check{p}_{ens} = \min_{\forall i} \left(p_{ens,i}\right).$$

Note that if Equation (16) holds for $\check{p}_{pro}$ and $\check{p}_{ens}$, it will also hold for any $p_{pro,i}$ and $p_{ens,i}$. We obtain the following value for $\check{p}_{ens}$ assuming $2 \times \check{p}_{pro} = \check{p}_{ens}$, i.e., that knowledge propagation is done twice as frequently as any ensemble membership check[5]:

$$\check{p}_{ens} \geq 6.88\,\text{ms},$$

and, hence, $\check{p}_{pro}$ has to be greater or equal to 3.44ms. Note that there cannot be a $p_{ens,i}$ that is less than $\check{p}_{ens}$. Similarly, $p_{pro,i}$ is bounded from below by $\check{p}_{pro}$. Otherwise, Equation (16) will not hold.

On the other hand, the upper bounds $\hat{p}_{ens}$ and $\hat{p}_{pro,i}$ need to fulfill the system's feasibility condition. That is, DEECo's closed-loop delay must be at most equal to the timing constraint $t_{1\text{m}}$:

$$D_{max} \leq t_{1\text{m}}. \tag{17}$$

D_{max} is DEECo's closed-loop delay for our case study as per Equation (2), i.e., where $R_{max} = r_{ICS} + r_{car}$ and $C_{max} = c_{ICS} + c_{car}$ in general expression $\bar{D}_{max}$ as given in Equation (3).

We choose $\hat{p}_{ens} = 14\,\text{ms}$ – twice as much as $\check{p}_{ens}$ – and hence $\hat{p}_{pro} = 7\,\text{ms}$, i.e., we again have $2 \times \hat{p}_{pro} = \hat{p}_{ens}$. With these values of $\hat{p}_{ens}$ and $\hat{p}_{pro}$, we verify next whether Equation (17) can be met. If not, new values of $\hat{p}_{ens}$ and $\hat{p}_{pro}$ need to be chosen – clearly, these should be greater than or equal to their lower bounds $\check{p}_{ens}$ and $\check{p}_{pro}$ respectively.

To test whether Equation (17) holds for the chosen $\hat{p}_{ens}$ and $\hat{p}_{pro}$, we need to compute the corresponding r_{ICS}, r_{car}, c_{ICS}, and c_{car}. We can compute r_{ICS} using Equation (6) and assuming that all processes are respectively released either at a $\hat{p}_{ens}$ or a $\hat{p}_{pro}$ rate[6].

$$r_{ICS} = 14\,\text{ms} \times \left(\frac{0.025\,\text{ms}}{7\,\text{ms}} + 68 \cdot \frac{2 \times 0.025\,\text{ms}}{14\,\text{ms}}\right) + 0.025\,\text{ms} + 68 \cdot (2 \times 0.025\,\text{ms}) \approx 7\,\text{ms}. \tag{18}$$

Similarly, we can compute r_{car} using Equation (7). In the car, there are only one ensemble process, one process to update the speed with the new one assigned by the ICS, and a knowledge propagation process. Again, we assume the ensemble process triggers the knowledge exchange process at cars. Assuming again $e_i = 25\,\mu\text{s}$, we obtain:

$$r_{car} = 14\,\text{ms} \times \left(\frac{0.025\,\text{ms}}{7\,\text{ms}} + \cdot \frac{2 \times 0.025\,\text{ms}}{14\,\text{ms}}\right) + (0.025\,\text{ms} + 2 \times 0.025\,\text{ms} = 0.18\,\text{ms}.$$

Now we need compute $\hat{c}_E$, i.e., the transmission time on the Ethernet link taking contention into account, using Equation (11):

$$\hat{c}_E = \hat{p}_{pro} \times 68 \times \frac{1.168\,\mu\text{s}}{\hat{p}_{pro}} + 68 \times 1.168\,\mu\text{s} = 158.85\,\mu\text{s},$$

[5]This is a design decision that needs to be taken. In general, since ensemble membership checks rely on local knowledge, it is meaningful that knowledge be updated as often as necessary to guarantee desired functionality.

[6]Note that processes with different rates are also possible; however, this is not meaningful in the context of our case study, where each process stands for an approaching car at the intersection.

where c_i and z_i have been replaced by c_E and $\hat{p}_{pro}$ respectively. Further, b_i is zero according to Equation (9), since we consider the lowest priority message for $\hat{c}_E$, i.e., the one suffering the most contention by other messages. The communication delay from and to the ICS, can be computed using Equation (12) and Equation (13):

$$c_{ICS} = 18.02\,\mu\text{s}, c_{car} = 333.38\,\mu\text{s}.$$

Finally, from Equation (2), we have that:

$$D_{max} = 2 \times 7\,\text{ms} + 2 \times 14\,\text{ms} + 0.3334\,\text{ms} + 0.0181\,\text{ms} + 7\,\text{ms} + 0.18\,\text{ms} \approx 50\,\text{ms},$$

which is less than $t_{1\text{m}} = 72ms$ – see Equation (14). That is, our ICS is able to meet all deadlines in the worst case.

7. Robustness under Unreliable Communication

In general, if many consecutive packets are lost on the communication channel, the system will experience malfunction putting safety into risk. In this section, we will determine the number of consecutive packets that can be lost at maximum without compromising safety. In other words, we quantify the system's *robustness* under unreliable communication.

As discussed above, Equation (2) states the worst-case delay incurred by our DEECo-based ICS in case of a fully reliable communication. This is obtained considering that data at a car can be updated immediately after knowledge has been propagated – for the reason that processes updating and propagating data are not synchronized with each other. As a result, this data will be sent the next time a knowledge propagation is performed, i.e., $\hat{p}_{pro}$ time later. If now this packet is lost on the communication channel, data will incur an additional delay equal to $\hat{p}_{pro}$. Further, if k_{car} denotes the number of consecutive packets that are lost, then data from the car to the ICS incurs the following delay:

$$\hat{p}_{pro} + k_{car} \times \hat{p}_{pro} + c_{car}, \tag{19}$$

where again c_{car} denotes the delay on the communication channel from the car to the ICS.

In a similar manner, if k_{ICS} denotes the number of consecutive packets that are lost from the ICS to the car, then data from the ICS incurs following delay to reach the car:

$$\hat{p}_{pro} + k_{ICS} \times \hat{p}_{pro} + c_{ICS}, \tag{20}$$

where, as discussed above, c_{ICS} is delay on the communication channel from the ICS to the car.

Let us denote by $k = k_{car} + k_{ICS}$ the total number of packets lost between the ICS and the car. We can now combine Equation (19) and Equation (20) to determine the closed-loop delay of the system in case of unreliable communication:

$$\hat{D}_{max} = (2 + k) \times \hat{p}_{pro} + 2 \times \hat{p}_{ens} + c_{ICS} + c_{car} + r_{ICS} + r_{car}, \tag{21}$$

where again r_{ICS} and r_{car} denote the maximum delay to finish computation at the ICS and the car respectively. Note that Equation (21) reduces to Equation (2) for $k = 0$, i.e., when no packets are lost on the communication channel.

Using the values of $\hat{p}_{pro}$, $\hat{p}_{ens}$, c_{car}, c_{ICS}, r_{car}, and r_{ICS} computed in the previous section, we can now determine the maximum k that can be tolerated without affecting the system's functionality and safety. That is the maximum k that makes $\hat{D}_{max}$ be at most equal to $t_{1\text{m}}$ as per

Equation (14). We denote this maximum k by k_{max}:

$$k_{max} = \left\lfloor \frac{t_{1\mathrm{m}} - D_{max}}{\hat{p}_{pro}} \right\rfloor = 3. \tag{22}$$

Equation (22) indicates that the sum of k_{car} and k_{ICS}, each of which represents the number of consecutive packets being lost in one or the other direction, cannot be more than 3 for the system to operate correctly.

Safety Mechanisms. Note that we can use the previous results to implement safety mechanisms at the ICS and at cars. In particular, whenever communication is lost for longer than $t_{1\mathrm{m}}$ time between the ICS and any car in the system or vice versa, both the ICS and the car switch to manual mode, i.e., the ICS starts working as standard traffic lights.

This can be implemented by DEECo processes that run at the different cars and at the ICS and trigger the manual mode in a decentralized manner. In other words, these processes behave as *watchdog timers* at the different nodes. They force a switch to manual mode at the corresponding node, if no packets have been received for longer than $t_{1\mathrm{m}}$ time. Note that here, for ease of exposition, we neglect the time which is necessary to process data at cars, i.e., $\hat{p}_{ens} + r_{car}$, and at the ICS, i.e., $\hat{p}_{ens} + r_{ICS}$. Whereas there is only one such watchdog processes at a car, there are multiple ones at the ICS; one for each car in the system.

It should be noticed that the ICS does not need to notify cars whenever it switches to manual mode; it suffices if it stops assigning speeds to them and cars themselves will automatically switch to manual mode. In the same way, if a car first switches to manual mode, the ICS will detect this on its own without need for notification from the car.

In the worst case, since we do not know which packets may be lost, we may have up to $3 \times t_{1\mathrm{m}}$ delay for the whole system, i.e., all cars and the ICS, to switch to manual mode in a decentralized manner. This results from considering the following conditions:

i) A packet from ICS is sent to arrive exactly $t_{1\mathrm{m}}$ time after its last packet at a given car.

ii) This packet from the ICS is lost at the communication channel such that the car (locally) switches to manual mode.

iii) All packets from the car to the ICS also get lost such that the ICS realizes that the car is in manual mode – and triggers itself a switch to manual mode – not until $t_{1\mathrm{m}}$ time later.

iv) All packets from the ICS to the remaining cars get lost such that, in the worst case, all other cars switch to manual mode $t_{1\mathrm{m}}$ time after the ICS.

As discussed above, this delay corresponds to 3 m in the trajectory of a car in our case study. Hence the ICS has to assign speeds to cars – in the automatic mode – such that there is sufficient distance between them taking vehicle dynamics into account (e.g., if a car suddenly breaks, it will not stop immediately due to its inertia, etc.).

Finally, only the ICS can decide to go back to the automatic mode whenever communication to all cars has normalized. To this end, the ICS needs to notify or start assigning speeds to all cars in the system. Note that the delay for switching to the automatic mode is given by $t_{1\mathrm{m}}$ since we assume a normal communication.

Priority levels	7
Message length	1024 bits
Packet send interval	7 ms ($\hat{p}_{pro}$ from the analysis)
ICS response delay	$\hat{p}_{pro} + \hat{p}_{ens} + r_{ICS} = 28$ ms
A car response delay	$\hat{p}_{pro} + \hat{p}_{ens} + r_{car} = 21.18$ ms
Bandwidth (Car to AP)	100 Mbps
Bandwidth (ICS to AP)	1 Gbps

Table 1: Simulation parameters

8. Experimental Evaluation

In this section, we validate the analysis presented in Section 6 by means of simulation. To this end, we created an OMNet++ simulation [29] using *INET* hardware models.

We have set up an OMNet++ simulation by manually implementing DEECo components as OMNet++ modules. In particular, we have implemented an OMNet++ module for each vehicle at the intersection and for the ICS. While the ICS is stationary, vehicles and their corresponding modules in OMNet++ move with given speeds. The modules generate network traffic that emulates the communication of vehicles entering and exiting the ICS's region of influence. This reflects the knowledge/data propagation for our DEECo-based ICS, from which we collect end-to-end communication latencies for a large set of simulated packet transmissions.

Our network topology consists of one ICS host connected by a full-duplex switch to three AP – see Figure 6. Vehicles connect dynamically to the AP adjusting message priorities as they get closer to the intersection. The communication from the switch to the ICS host is performed under message prioritization according to the Ethernet 802.1Q standard. Our simulation scenario spans different numbers of vehicles (20, 50 and 70 correspondingly) exchanging packets with the ICS. Table1 summarizes the most important simulation parameters considered in our evaluation.

Figure 7 and Table 2 show the results of our simulation with respect to closed-loop reaction time – i.e., the Car-ICS-Car delay – and for an increasing number of consecutive packet losses at the communication channel. In the case that no packets are lost, this figure shows that our $D_{max} = 50$ ms – computed at the end of Section 6 – is safe. That is, in this case, all delay values in the system are always less than 50 ms even for 70 cars, i.e., two more cars than what it is

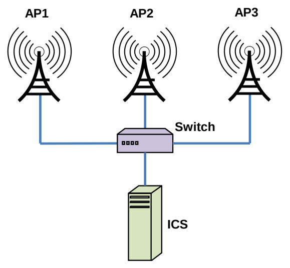


Figure 6: Simulated network consisting of three access points (APs) and a switch

considered and allowed by the analysis presented in the above sections.

Evaluation under Unreliable Communication. Now, we discuss our simulation results for a varying number of consecutive packet losses either from the car to the ICS or from the ICS to the car. As it can be observed in Figure 7, the system operates properly – i.e., the Car-ICS-Car delay is below below $t_{1\mathrm{m}} = 72\,\mathrm{ms}$ – for up to 3 consecutive packet losses, which validates our analysis in Section 7. Clearly, the more packets are lost, the higher the Car-ICS-Car delay is; however, this is always less than the computed threshold $t_{1\mathrm{m}}$ and, hence, the system can remain in automatic mode.

For the case of 4 packets lost, also depicted in Figure 7, the Car-ICS-Car delay starts exceeding the threshold $t_{1\mathrm{m}} = 72\,\mathrm{ms}$ – even when considering only 20 cars at the intersection. As a result, the system cannot tolerate more than 3 consecutive packet losses without switching to manual mode. This again is in accordance with the computed upper bound on packet losses given in Equation (22).

Realism of the Evaluation. The above results are based on a simulation and, thus, they may differ in reality. In particular, we have made a number of assumptions which may not hold and, hence, have an impact on our evaluation results. In the following, we discuss this in more detail.

- The computed $t_{1\mathrm{m}}$ may not hold. This is based on the assumption that cars/vehicles can have speeds of up to 50 Km/h – see (14). However, since cars are responsible for their speeds constant, in reality, it may happen that one or more cars exceed this speed limit by some amount. A solution to this is to consider a safety margin and, for example, compute a new $t_{1\mathrm{m}}$ for 60 Km/h instead. However, it now may happen that the ICS cannot meet this deadline anymore. To overcome this problem, the number of cars at the intersection can be restricted to a safe value. If more cars than safe enter the ICS's region of influence, it will switch to manual mode. Clearly, this higher speed limit can also be exceeded, however, this would fall into malicious behavior and, hence, the ICS would also switch to manual mode.
- The computed maximum number of cars at the intersection n may also not hold. This is based on an assumption on the minimum length of cars and on the maximum possible distance between any two cars at the intersection – see (15). If these assumptions do not hold in practice, the maximum number of cars at the intersection may potentially increase. This has impact on the WCRT of the ICS r_{ICS} and on the worst-case communication delay from a car to the ICS c_{car}. As a result, the ICS may probably not be able to meet deadlines anymore and, hence, it will have to switch to manual mode, if more cars than expected enter its region of influence.
- The WCET of processes at the cars and at the ICS may be greater than the assumed $e_i = 50\,\mu\mathrm{s}$. This will have direct impact on the WCRT at the car r_{car} and at the ICS r_ICS. As a consequence, the ICS may not be able to meet deadlines anymore and, again, it will have to switch to manual mode, if a given number of cars is exceeded at the intersection.
- The bandwidths assumed for the different segments (either from the car to the AP or from the AP to the ICS)are less than those assumed in Table 1. This leads to increased communication delays in both directions from the car to the ICS and vice versa. The ICS may stop being able to meet deadlines and, again, will have to restrict the number of cars at the intersection.

From the above discussion, it should be clear that we can account for discrepancies between our simulated and a real-life ICS by taking a conservative estimate on the maximum number of cars that the ICS can handle. If, in practice, this number is exceed, the ICS will switch to manual mode.

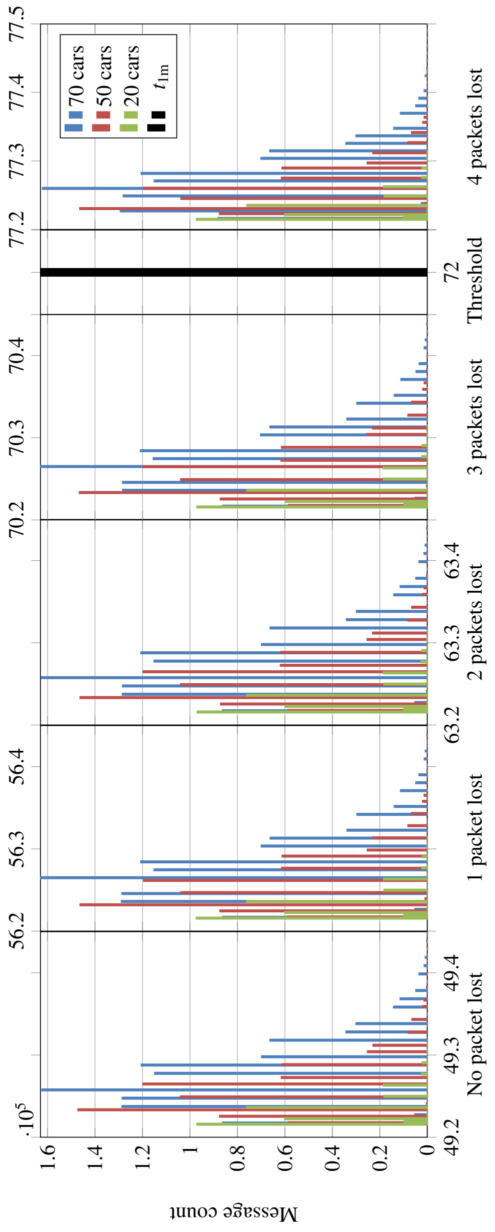


Figure 7: Car-ICS-Car closed-loop reaction times in milliseconds

No packet lost	20 vehicles	50 vehicles	70 vehicles
Mean	49.2245	49.2507	49.2694
Std. Dev.	0,0157	0.00298	0.03723
Variance	$2.42 \cdot 10^{-7}$	$8.88 \cdot 10^{-7}$	$13.86 \cdot 10^{-7}$
1st Quartile	49.2124	49.2315	49.2449
Median	49.2182	49.2449	49.2582
3rd Quartile	49.2315	49.2716	49.285
Max	49.3117	49.4453	49.5121
1 packet lost			
Mean	56.2245	56.2507	56.2692
Std. Dev.	0,0155	0.00299	0.0371
Variance	$2.41 \cdot 10^{-7}$	$8.93 \cdot 10^{-7}$	$13.8 \cdot 10^{-7}$
1st Quartile	56.2111	56.2315	56.2449
Median	56.2182	56.2449	56.2582
3rd Quartile	56.2315	56.2716	56.285
Max	56.3117	56.4319	56.4987
2 packets lost			
Mean	63.2246	63.2508	63.2694
Std. Dev.	0,0156	0.00298	0.0372
Variance	$2.441 \cdot 10^{-7}$	$8.91 \cdot 10^{-7}$	$13.9 \cdot 10^{-7}$
1st Quartile	63.2124	63.2315	63.2449
Median	63.2182	63.2449	63.2582
3rd Quartile	63.2315	63.2716	63.285
Max	63.3117	63.4453	63.5121
3 packets lost			
Mean	70.2246	70.2509	70.2693
Std. Dev.	0,0156	0.00298	0.0371
Variance	$2.435 \cdot 10^{-7}$	$8.9 \cdot 10^{-7}$	$13.75 \cdot 10^{-7}$
1st Quartile	70.2124	70.2315	70.2449
Median	70.2182	70.2449	70.2582
3rd Quartile	70.2315	70.2716	70.285
Max	70.3117	70.4453	70.4987
4 packets lost			
Mean	77.2245	77.2507	77.2694
Std. Dev.	0,0156	0.00298	0.0372
Variance	$2.418 \cdot 10^{-7}$	$8.9 \cdot 10^{-7}$	$13.85 \cdot 10^{-7}$
1st Quartile	77.2124	77.2315	77.2449
Median	77.2182	77.2449	77.2582
3rd Quartile	77.2315	77.2716	77.285
Max	77.3117	77.4319	77.5388

Table 2: Reaction time statistics (values given in milliseconds)

9. Concluding Remarks

In this paper, we have presented DEECo as a special-purpose, component-based, design and development framework for open-ended CPS. DEECo specifically targets at dynamic distributed systems and, thus, provides systematic software engineering mechanisms to describe and analyze such complex application scenarios. These mechanisms mainly consist in modeling transitory interactions between one or more components in the system.

We extended DEECo's design flow by a technique to estimate worst-case, closed-loop, response times between DEECo components. This effectively allows guaranteeing real-time requirements from high-level DEECo-based designs provided that the underlying platform supports real-time, e.g., real-time OS, priority-based communication protocols, etc. Clearly, if the underlying technologies are nondeterministic, then it is not possible to provide any timing guarantees and, as a consequence, no safety-critical applications can be implemented on their basis.

We illustrated our proposed technique based on an intelligent crossroad scenario. Towards this, we derived the worst-case delay D_{max} of a DEECo-based system – see Equation (2). This analysis is general enough and can be used for other applications. Note that the term $2 \times \hat{p}_{pro} + 2 \times \hat{p}_{ens}$ is the overhead by DEECo, whereas $c_{ICS} + c_{car}$ and $r_{ICS} + r_{car}$ stand for the communication and the computation overhead respectively. DEECo's overhead is configurable by properly choosing $\hat{p}_{pro}$ and $\hat{p}_{ens}$ which again need to be in accordance with the application requirements. The communication and computation overhead will depend on the used technologies such as communication protocols, scheduling algorithms, etc.

Based on our analysis, we evaluated the robustness of a DEECo-based design against packet losses at the communication channel. Towards this, we analytically obtained a upper bound on the number of packets that can be lost without affecting the system's safety. We further validated this bound by means of extensive experiments based on an OMNet++ simulation. In addition, we proposed and discussed safety mechanisms that can be integrated into a DEECo design in order to adapt to unpredicted communication loss between components.

We envision integrating the proposed technique into the existing ensemble development life cycle [30], which provides a systematic approach (i.e., methodology) towards engineering open-ended CPS. The presented work fits into the modeling part of that cycle, which is followed by verification performed on the basis of simulation techniques – similar to the procedure shown in this paper. An important aspect is also the requirements engineering part, which should besides functional properties also account for extra-functional, in particular, real-time aspects.

Overall, the technique presented in this paper allows reasoning about real-time requirements at the component level and constitutes a necessary step towards holistic software engineering methods for modern cyber-physical systems.

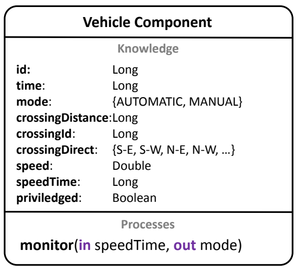
Acknowledgments

This work was partially supported 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.

Appendix A. DEECo-based ICS implementation

In the following, we show an example of a DEECo model for the ICS case study. The study is based on the CDEECo (C implementation of DEECo) library which provides scheduling via

FreeRTOS[7], deployment on the STM32F4[8] board, communication via the IEEE802.15.4 interface and mapping of DEECo ensembles and components into the C++ language. The sources of the case study draft and network traffic simulation are placed on the GitHub[9]. The main purpose of the published code is to displays the way how DEECo application can be modeled as real-time embedded application written in C++. In order to navigate in the repository please see the README file.



Vehicle Component

Knowledge

id:	Long
time:	Long
mode:	{AUTOMATIC, MANUAL}
crossingDistance:	Long
crossingId:	Long
crossingDirect:	{S-E, S-W, N-E, N-W, …}
speed:	Double
speedTime:	Long
priviledged:	Boolean

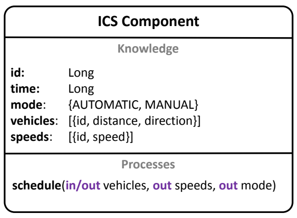
Processes

monitor(in speedTime, out mode)

Figure A.1: Vehicle component specification

DEECo components. Figure A.1 specifies the Vehicle component that is characterized by the set of attributes (together with their types) listed in the figure and the following process:

- *monitor(in speedTime, out mode)*
  The process is responsible for monitoring whether input data is obsolete or not (i.e., the time of the last speed update must be less than a *threshold* given by our real-time analysis in Section 6). If so, i.e., if the value from *ICS* is obsolete, then *mode* is set to the MANUAL. As input parameter, it takes the *ICS* identifier, whereas it returns the vehicle's *mode*.



ICS Component

Knowledge

id:	Long
time:	Long
mode:	{AUTOMATIC, MANUAL}
vehicles:	[{id, distance, direction}]
speeds:	[{id, speed}]

Processes

schedule(in/out vehicles, out speeds, out mode)

Figure A.2: ICS component specification

Figure A.2 specifies the ICS component that is characterized by the set of attributes (together with their types) listed in the figure and the following process:

[7]http://www.freertos.org/
[8]http://www.st.com/stm32f4
[9]https://github.com/d3scomp/ICS-CDEECo

27

- *schedule(in/out vehicles, out arrivals, out mode)*
  This process is responsible for computing and monitoring the speeds of approaching vehicles/cars depending on the current traffic situation. Towards this, given cars' current speeds and directions, it computes the time at which they reach the intersection. Taking cars' lengths and widths into account, it adjusts their speeds to avoid conflicts. Which car is allowed to cross first depends on which order they arrive at ICS's region of influence and on whether they are privileged or not. If the ICS detects that a car or vehicle does not respect the assigned speed, it changes the value of *mode* to MANUAL, i.e., it starts working as standard traffic lights. The same happens, if communication is lost to one or more cars. As input parameters, this process takes *vehicles*, i.e., a collection of the most recent states of cars/vehicles at the crossing. As output parameters, it returns *vehicles*, where the *speed* attribute of each vehicle is updated, and *mode*.

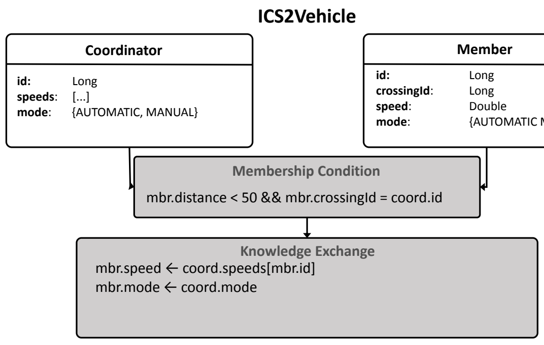


Figure A.3: ICS2VehicleEnsemble specification

Figure A.3 and Figure A.4 show the specification of ensembles, i.e., interactions or relations, between the ICS and Vehicle components. The attributes of the *coordinator* in these ensembles match the attributes of the ICS component, while the *member's* attributes match the Vehicle component. Note that, in the end, this is the same interaction/relation but specified from the *perspective* of the member – see Figure A.3 – and of the coordinator – see Figure A.4.

[1] K. Beetz, W. Böhm, Challenges in engineering for software-intensive embedded systems, in: Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology, Springer, 2012.
[2] AUTOSAR: Layered software architecture, `http://autosar.org/download/R4.0/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf`.
[3] W. Reisig, Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies, Springer, 2013.
[4] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, F. Plasil, DEECo: An ensemble-based component system, in: Proceedings of the International ACM Sigsoft Symposium on Component-based Software Engineering (CBSE), 2013.
[5] F. Baude, A perspective on the CoreGRID grid component model, in: Euro-Par 2011: Parallel Processing Workshops, Springer, 2012.
[6] A. Masrur, M. Kit, T. Bures, W. Hardt, Towards component-based design of safety-critical cyber-physical applications, in: Proceedings of the Euromicro Conference on Digital Systems Design (DSD), 2014.
[7] J. Keznikl, T. Bures, F. Plasil, I. Gerostathopoulos, P. Hnetynka, N. Hoch, Design of ensemble-based component systems by invariant refinement, in: Proceedings of the International ACM Sigsoft Symposium on Component-based Software Engineering (CBSE), 2013.

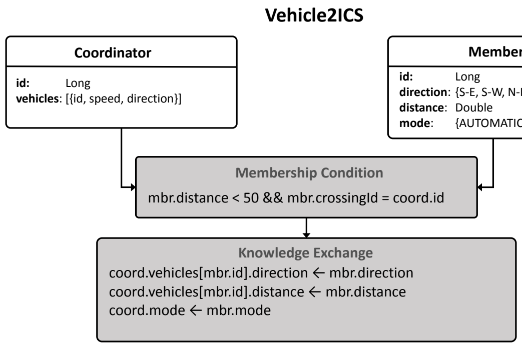

Vehicle2ICS
Coordinator
id: Long
vehicles: [{id, speed, direction}]
Member
id: Long
direction: {S-E, S-W, N-E, …}
distance: Double
mode: {AUTOMATIC, MANUAL}
Membership Condition
mbr.distance < 50 && mbr.crossingId = coord.id
Knowledge Exchange
coord.vehicles[mbr.id].direction ← mbr.direction
coord.vehicles[mbr.id].distance ← mbr.distance
coord.mode ← mbr.mode

Figure A.4: Vehicle2ICSEnsemble specification

[8] C++ implementation of the DEECo runtime environment, available online: `http://github.com/d3scomp/CDEECo` (2014).

[9] M. Kit, I. Gerostathopoulos, T. Bures, P. Hnetynka, F. Plasil, An Architecture Framework for Experimentations with Self-Adaptive Cyber-Physical Systems, in: International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), IEEE, 2015.

[10] S. Zeadally, R. Hunt, Y.-S. Chen, A. Irwin, A. Hassan, Vehicular ad hoc networks (VANETS): status, results, and challenges, Telecommunication Systems 50 (4).

[11] K. Klobedanz, C. Kuznik, A. Thuy, W. Mueller, Timing modeling and analysis for AUTOSAR-based software development - a case study, in: Proceedings of Conference on Design, Automation, and Test in Europe (DATE), 2010.

[12] O. Sokolsky, A. Chernoguzov, Performance analysis of AADL models using real-time calculus, in: Foundations of Computer Software: Future Trends and Techniques for Development, Springer, 2010.

[13] L. Thiele, S. Chakraborty, M. Naedele, Real-time calculus for scheduling hard real-time systems, in: Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), 2000.

[14] A. Basu, M. Bozga, J. Sifakis, Modeling heterogeneous real-time components in BIP, in: Proceedings of the IEEE International Conference on Software Engineering and Formal Methods (SEFM), 2006.

[15] H. Espinoza, D. Cancila, B. Selic, S. Gerard, Challenges in combining SysML and MARTE for model-based design of embedded systems, in: Model Driven Architecture - Foundations and Applications, Springer, 2009.

[16] S. Becker, H. Koziolek, R. Reussner, The palladio component model for model-driven performance prediction, Systems and Software 82 (1).

[17] C. Etzien, T. Gezgin, S. Froschle, S. Henkler, A. Rettberg, Contracts for evolving systems, in: IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013.

[18] T. Gezgin, C. Etzien, Correct by prognosis: Methodology for a contract-based refinement of evolution models, in: Complex Systems Design & Management (CSD&M), 2014.

[19] T. Gezgin, S. Henkler, A. Rettberg, I. Stierand, Contract-based compositional scheduling analysis for evolving systems, in: G. Schirner, M. Gtz, A. Rettberg, M. Zanella, F. Rammig (Eds.), Embedded Systems: Design, Analysis and Verification, Vol. 403 of IFIP Advances in Information and Communication Technology, 2013.

[20] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, F. Plasil, Gossiping components for cyber-physical systems, in: P. Avgeriou, U. Zdun (Eds.), Software Architecture, Vol. 8627 of Lecture Notes in Computer Science, Springer International Publishing, 2014.

[21] C. Liu, J. Layland, Scheduling algorithms for multiprogramming in hard real-time environments, Journal of the Association for Computing Machinery 20 (1).

[22] N. Audsley, A. Burns, M. Richardson, K. Tindell, A. Wellings, Applying new scheduling theory to static priority pre-emptive scheduling, Software Engineering Journal 8 (5).

[23] N. Shah, F. Bastani, S. Kumar, I.-L. Yen, Real-time car-to-car communication protocol for intersecting roads, in: Proceedings of the International Conference on ITS Telecommunications (ITST), 2008.

[24] N. Shah, S. Kumar, F. Bastani, I.-L. Yen, Optimization models for assessing the peak capacity utilization of intelli-

gent transportation systems, European Journal of Operational Research 216 (1).
[25] S.-Y. Pyun, H. Widiarti, Y.-J. Kwon, D.-H. Cho, J.-W. Son, TDMA-based channel access scheme for V2I communication system using smart antenna, in: Proceedings of the IEEE Conference on Vehicular Networking (VNC), 2010.
[26] S.-Y. Pyun, H. Widiarti, Y.-J. Kwon, J.-W. Son, D.-H. Cho, Group-based channel access scheme for a V2I communication system using smart antenna, IEEE Communications Letters 15 (8).
[27] IEEE 802.11p standard: Wireless LAN MAC and PHY specifications amendment 6: Wireless access in vehicular environments, `http://standards.ieee.org/findstds/standard/802.11p-2010.html`.
[28] IEEE 802.1Q standard: LANs and WANs – MAC bridges and virtual bridged LANs, `http://standards.ieee.org/findstds/standard/802.1Q-2011.html`.
[29] A. Varga, R. Hornig, An overview of the OMNeT++ simulation environment, in: Proceedings of the International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTOOLS), 2008.
[30] T. Bures, R. D. Nicola, I. Gerostathopoulos, N. Hoch, M. Kit, N. Koch, G. V. Monreale, U. Montanari, R. Pugliese, N. Serbedzija, M. Wirsing, F. Zambonelli, A life cycle for the development of autonomic systems: The e-mobility showcase, in: Proceedings of the Workshop on Challenges for Achieving Self-Awareness in Autonomic Systems (AWARENESS), 2013.

Chapter 5

Conclusions & Open Challenges

The work presented in this thesis focuses on delivering a realization platform for the DEECo component model. In a bigger frame, it contributes to the SCPS software engineering process, by simplifying and automating both the development and deployment stages of that process. The resulting jDEECo platform provides (i) Java implementation constructs that map both DEECo components and ensembles, (ii) a runtime environment, which automates deployment and execution of components and ensembles as well as data coordination between deployment units, and (iii) the jDEECoSim - a simulation tool allowing for system validation with respect to its future real-life deployment. As the contribution of the thesis goes to all of those points, the biggest effort has been put into (ii) and, in particular, the distributed data coordination part. As a result, the jDEECo platform supports heterogeneous deployment environments, including infrastructure-based and infrastructure-less networks. The implemented techniques have been validated on multiple scenarios that exemplified SCPS usage in real-life settings.

As such, with respect to the research goals formulated in Section 1.3, the goal **G1** has been achieved by proposing the DEECo component model being the result of the collaborative work conducted within the author's research group.

The goal **G2** (together with its sub-goals **G2a** and **G2b**) is addressed by delivering (i) and (ii). In particular, techniques introduced in Sections 4.3 and 4.5, designed to optimize network utilization, address **G2b**.

Similarly, the goal **G3**, which speaks about the validation stage of a developed SCPS, has been achieved by delivering (iii).

With respect to the proof of the concept, the jDEECo platform has been validated on multiple use-case scenarios (some of them incorporated across publications presented in Section 4). All of them, were based on real-life examples of SCPS systems, stemming from different domains like catastrophe recovery [42], automotive industry [9] and cloud computation [9]. The jDEECo platform, in its current shape, is a result of many prototype implementations and iterations of the idea, which finally brought it to the state where it can be successfully used not only for SCPS development but also for other purposes (e.g. agent-based system simulation).

In terms of future challenges, there is a wide range of possibilities that could potentially contribute to the domain of SCPS development. Focusing on the DEECo component model, an interesting research area is the component self-adaptation, with respect to different contexts it executes in, using the idea of **roles**. As DEECo is designed to deliver means for SCPS development, there is not an explicit tool that would facilitate adaptive behavior modeling. The idea of a component role has already been brought up, in Section 2.1.3.1, describing the HELENA approach. It specifies a component role as a dynamically adopted feature, encapsulating different behaviors that potentially could form an inheritance hierarchy. Implementation (and further elaboration) of the idea of roles in the scope of the DEECo component model could simplify inclusion of adaptation techniques into the DEECo-based SCPS design process.

The idea of roles is related to another interesting topic, which is **security**. Currently in the DEECo component model, apart from the idea of component interfaces (limiting the view on component attributes), there are no proper means for implementing secure SCPS. An example approach that could possibly be reused (or adopted), for needs of the DEECo component model, is given by policies proposed in SCEL (see Section 2.1.3.1). In SCEL, polices coexists together with components and manage access to their data. This, applied in the context of DEECo, would allow for reasoning about component interaction in terms of permissions and privileges yet during the design time of an SCPS. An additional aspect that goes rather to the jDEECo platform is component data encryption. Some initial work related to that topic has already been done by implementing into jDEECo a possibility for restricting access to component data and adding encryption for component data exchanged over the network [Stu15]

Another challenge, which refers to early-stage analyzes of a system model, goes to **data consistency**. Taking into account that communication in case of SCPS is based mainly on unreliable wireless network infrastructure, which is prone to interference and delays in data transmission, it may happen that components deployed on different network nodes have a different view on the same part of the system. Therefore, having a tooling, which would allow for assessment of possible discrepancies in component data, could forecast potential implications on system behavior yet at the design phase of the SCPS development process. The research work in this area has already been initiated and some of its results can be found in [AABG+14a].

Data inconsistencies are also an issue at the jDEECo platform level. Currently, whenever a component data is transmitted over the network, it is first divided into packets complying with the low-level communication protocol requirements (i.e. packet maximum size). It is obvious that packets could be dropped or corrupted during the transmission process. Currently, however, the jDEECo runtime implements a simple approach, where data is assembled based on the order packets are received and disregards the sender identity and to which stream packets belong. In cases where data semantics matters, this may be problematic, as depending on particular use-case scenarios some parts of data are strongly related to each other. As a simple example, geographical coordinates can be considered. It would not make much sense to send separately latitude and longitude, as one could change during the transmission process of other, leading to inconsistencies across component data. Thus, they need to be transmitted together to ensure data consistency.

Currently, the jDEECo platform lacks any support for automated division of data into packets consisting semantically related information, which would definitely improve the overall system reliability.

References

[AABG+14a] R. Al Ali, T. Bures, I. Gerostathopoulos, J. Keznikl, and F. Plasil. Architecture adaptation based on belief inaccuracy estimation. In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pages 87–90, April 2014.

[AABG+14b] R. Al Ali, T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil. DEECo: an Ecosystem for Cyber-Physical Systems. In *ICSE '14: Companion Proceedings of the 36th International Conference on Software Engineering*, pages 610–611. ACM, June 2014. Poster and extended abstract. Available online: http://-d3s.mff.cuni.cz/publications/.

[AABG+14c] 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.

[ABH+06] N. Alechina, R. H. Bordini, J. F. Hubner, M. Jago, and B. Logan. Belief revision for AgentSpeak agents. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '06, pages 1288–1290, New York, NY, USA, 2006. ACM.

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *Computer*, 19(8):26–34, August 1986.

[AIM10] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, October 2010.

[ARS15] P. Arcaini, E. Riccobene, and P. Scandurra. Modeling and analyzing MAPEK feedback loops. In *SEAMS'15: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, May 2015.

[BBD+06] R. H. Bordini, L. Braubach, M. Dastani, A. El, F. Seghrouchni, J. J. Gomez-sanz, J. Leite, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems, 2006.

[BCC+08] T. Bures, J. Carlson, I. Crnkovic, S. Sentilles, and A. V. Feljan. Procom - the progress component model reference manual, version 1.0. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-230/2008-1-SE, June 2008.

[BCS04] E. Bruneton, T. Coupaye, and J. Stefani. *The Fractal Component Model*, February 2004. Version 2.0-3.

[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.

[Ber13] Bernd Leukert. Production of the future. How to prepare for the fourth industrial revolution. *SAP AG, e-Book*, November 2013.

[BFCA14] N. Bencomo, R. B. France, B. H. C. Cheng, and U. Aßmann, editors. *Models@run.time - Foundations, Applications, and Roadmaps [Dagstuhl Seminar 11481, November 27 - December 2, 2011]*, volume 8378 of *Lecture Notes in Computer Science*. Springer, 2014.

[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. Computational Model for Gossiping Components in Cyber-Physical Systems. Technical Report D3S-TR-2014-03, Department of Distributed and Dependable Systems, April 2014. Available online: http://d3s.mff.cuni.cz/publications/.

[BGH+14b] 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+15] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil. The Invariant Refinement Method. In *Software Engineering for Collective Autonomic Systems*, pages 405–428. Springer, 2015.

[BHK+14] T. Bures, V. Horky, M. Kit, L. Marek, and P. Tuma. Towards performance-aware engineering of autonomic component ensembles. 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 131–146. Springer Berlin Heidelberg, 2014.

[BHP06] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, pages 40–48, August 2006.

[Boc11] H. Bock. The OSGi framework. In *The Definitive Guide to NetBeans™ Platform 7*, pages 49–55. Apress, 2011.

[BPC+07] P. Baronti, P. Pillai, V. W. C. Chook, S. Chessa, A. Gotta, and Y. F. Hu. Wireless sensor networks: A survey on the state of the art and the 802.15.4 and ZigBee standards. *Comput. Commun.*, 30(7):1655–1695, May 2007.

[Bro13] M. Broy. Engineering cyber-physical systems: Challenges and foundations. In M. Aiguier, Y. Caseau, D. Krob, and A. Rauzy, editors, *Complex Systems Design & Management*, pages 1–13. Springer Berlin Heidelberg, 2013.

[BW98] A. Brown and K. Wallnau. The current state of CBSE. *Software, IEEE*, 15(5):37–46, September 1998.

[CCG+09] V. Cardellini, E. Casalicchio, V. Grassi, F. Lo Presti, and R. Mirandola. Towards self-adaptation for dependable service-oriented systems. In R. de Lemos, J.-C. Fabre, C. Gacek, F. Gadducci, and M. ter Beek, editors, *Architecting Dependable Systems VI*, volume 5835 of *Lecture Notes in Computer Science*, pages 24–48. Springer Berlin Heidelberg, 2009.

[CMMP09] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco. Tuple Space Middleware for Wireless Networks. In B. Garbinato, H. Miranda, and L. Rodrigues, editors, *Middleware for Network Eccentric and Mobile Applications*, pages 245–264. Springer Press, 2009. Invited contribution.

[CSS11] H. Cam, O. Sahingoz, and A. Sonmez. Wireless sensor networks based on publish/subscribe messaging paradigms. In J. Riekki, M. Ylianttila, and M. Guo, editors, *Advances in Grid and Pervasive Computing*, volume 6646 of *Lecture Notes in Computer Science*, pages 233–242. Springer Berlin Heidelberg, 2011.

[DGH+87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. ACM.

[DNFLP13] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese. A Language-Based Approach to Autonomic Computing. In B. Beckert, F. Damiani, F. Boer, and M. Bonsangue, editors, *FMCO 2011*, volume 7542 of *Lecture Notes in Computer Science*, pages 25–48. Springer Berlin Heidelberg, 2013.

[EHL07] C. Escoffier, R. Hall, and P. Lalanda. iPOJO: an extensible service-oriented component framework. In *Services Computing, 2007. SCC 2007. IEEE International Conference on*, pages 474–481, July 2007.

[ER94] E. Ephrati and J. S. Rosenschein. Divide and conquer in multi-agent planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1)*, AAAI '94, pages 375–380, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[Eri11] Ericsson. More than 50 billion connected devices. *Ericsson White Paper*, February 2011.

[Fis98] K. Fischer. An agent-based approach to holonic manufacturing systems. In L. Camarinha-Matos, H. Afsarmanesh, and V. Marik, editors, *Intelligent Systems for Manufacturing*, volume 1 of *IFIP — The International Federation for Information Processing*, pages 3–12. Springer US, 1998.

[Fis99] K. Fischer. Holonic multiagent systems. Theory and applications. In P. Barahona and J. Alferes, editors, *Progress in Artificial Intelligence*, volume 1695 of *Lecture Notes in Computer Science*, pages 34–48. Springer Berlin Heidelberg, 1999.

[FMF+12] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jezequel. A dynamic component model for cyber physical systems. In *CBSE '12: Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*, pages 135–144. ACM, 2012.

[FSS03] K. Fischer, M. Schillo, and J. Siekmann. Holonic multiagent systems: A foundation for the organisation of multiagent systems. In V. Mařík, D. McFarlane, and P. Valckenaers, editors, *Holonic and Multi-Agent Systems for Manufacturing*, volume 2744 of *Lecture Notes in Computer Science*, pages 71–80. Springer Berlin Heidelberg, 2003.

[GGHK09] N. Gaud, S. Galland, V. Hilaire, and A. Koukam. An organisational platform for holonic and multiagent systems. In K. Hindriks, A. Pokahr, and S. Sardina, editors, *Programming Multi-Agent Systems*, volume 5442 of *Lecture Notes in Computer Science*, pages 104–119. Springer Berlin Heidelberg, 2009.

[Gil97] D. Gilbert. Intelligent agents: The right information at the right time. *Technical Report IBM Corp., Research Triangle Park, NC*, page http://www.networkin, 1997.

[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, September 2014.

[GKG02] I. Gupta, A.-M. Kermarrec, and A. Ganesh. Efficient epidemic-style protocols for reliable and scalable multicast. In *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*, pages 180–189, 2002.

[GKM03] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. Comput.*, 52(2):139–149, February 2003.

[GSB12] J. Gozalvez, M. Sepulcre, and R. Bauza. Ieee 802.11p vehicle to infrastructure communications in urban environments. *Communications Magazine, IEEE*, 50(5):176–183, May 2012.

[GSC09] D. Garlan, B. Schmerl, and S.-W. Cheng. Software architecture-based self-adaptation. In Y. Zhang, L. T. Yang, and M. K. Denko, editors, *Autonomic Computing and Networking*, pages 31–55. Springer US, 2009.

[HC08] J. Hui and D. Culler. Extending IP to Low-Power, Wireless Personal Area Networks. *Internet Computing, IEEE*, 12(4):37–45, July 2008.

[HCeA03] D. Henriksson, A. Cervin, and K. erik Arzén. Truetime: Real-time control system simulation with MATLAB/Simulink. In *Proc. of the Nordic MATLAB Conference*, 2003.

[Hin09] K. Hindriks. Programmingrationalagents in goal. In A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini, editors, *Multi-Agent Programming:*, pages 119–157. Springer US, 2009.

[HK14] R. Hennicker and A. Klarl. Foundations for Ensemble Modeling–The Helena Approach. In *Specification, Algebra, and Software*, volume 8373 of *Lecture Notes in Computer Science*, pages 359–381. Springer Berlin Heidelberg, 2014.

[HPB+10] P. Hosek, T. Pop, T. Bures, P. Hnetynka, and M. Malohlava. Comparison of component frameworks for real-time embedded systems. In L. Grunske, R. Reussner, and F. Plasil, editors, *Component-Based Software Engineering*, volume 6092 of *Lecture Notes in Computer Science*, pages 21–36. Springer Berlin Heidelberg, 2010.

[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.

[IH08] T. Issariyakul and E. Hossain. *Introduction to Network Simulator NS2*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[JM96] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, pages 153–181. Kluwer Academic Publishers, 1996.

[JR06] C. Julien and G. Roman. EgoSpaces: facilitating rapid development of context-aware mobile applications. *Software Engineering, IEEE Transactions on*, 32(5):281–298, May 2006.

[KAE+06] T. Kosch, C. Adler, S. Eichler, C. Schroth, and M. Strassberger. The scalability problem of vehicular ad hoc networks and how to solve it. *Wireless Communications, IEEE*, 13(5):22–28, October 2006.

[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 *WICSA/ECSA '12: Proceedings of the Joint 10th Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture*, pages 249–252. IEEE Computer Society, August 2012.

[KCH15] 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, 2015.

[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. Available online: http://d3s.mff.cuni.cz/publications/.

[KGJ09] R. Kota, N. Gibbins, and N. R. Jennings. Self-organising agent organisations. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, pages 797–804, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[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, 2014.

[KPM+15] M. Kit, F. Plasil, V. Matena, T. Bures, and O. Kovac. Employing Domain Knowledge for Optimizing Component Communication. In *CBSE '15: Proceedings of the 18th International ACM Sigsoft Symposium on Component-based Software Engineering*. ACM, May 2015.

[Lam05] W. K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches (Prentice Hall Modern Semiconductor Design Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[Led15] L. Lednicki. *Software and Hardware Models in Component-based Development of Embedded Systems*. PhD thesis, Malardalen University, January 2015.

[LM11] R. Lanotte and M. Merro. Semantic analysis of gossip protocols for wireless sensor networks. In J.-P. Katoen and B. König, editors, *CONCUR 2011 – Concurrency Theory*, volume 6901 of *Lecture Notes in Computer Science*, pages 156–170. Springer Berlin Heidelberg, 2011.

[LMPT14] M. Loreti, A. Margheri, R. Pugliese, and F. Tiezzi. On programming and policing autonomic computing systems. 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 164–183. Springer Berlin Heidelberg, 2014.

[LPR10] J. Leitao, J. Pereira, and L. Rodrigues. Gossip-based broadcast. In X. Shen, H. Yu, J. Buford, and M. Akon, editors, *Handbook of Peer-to-Peer Networking*, pages 831–860. Springer US, 2010.

[LS10] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. Lee and Seshia, 1 edition, 2010.

[McE14] A. McEwen. *Designing the Internet of Things*. Wiley, 2014.

[Min98] N. Minar. Designing an ecology of distributed agents. Master's thesis, Massachusetts Institute of Technology, September 1998.

[MKBH14] A. Masrur, M. Kit, T. Bures, and W. Hardt. Towards component-based design of safety-critical cyber-physical applications. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 254–261, Aug 2014.

[MKM+16] A. Masrur, M. Kit, V. Matěna, T. Bureš, and W. Hardt. Component-based design of cyber-physical applications with safety-critical requirements. *Microprocessors and Microsystems*, 2016.

[ML15] E. Mercier-Laurent. *The Innovation Biosphere: Planet and Brains in the Digital Era*. Wiley, 2015.

[MPR06] A. L. Murphy, G. P. Picco, and G.-C. Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Software Engineering Methodologies*, 15(3):279–328, July 2006.

[Mul07] G. Mulligan. The 6LoWPAN architecture. In *Proceedings of the 4th workshop on Embedded networked sensors*, EmNets '07, pages 78–82, New York, NY, USA, June 2007. ACM.

[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.

[PBRD03] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental), July 2003.

[PC98] N. Pryce and S. Crane. A model of interaction in concurrent and distributed systems. In F. van der Linden, editor, *Development and Evolution of Software Architectures for Product Families*, volume 1429 of *Lecture Notes in Computer Science*, pages 57–65. Springer Berlin Heidelberg, 1998.

[PR99] C. Perkins and E. Royer. Ad-hoc on-demand distance vector routing. In *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA '99. Second IEEE Workshop on*, pages 90–100, Feb 1999.

[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.

[RBB+11] R. Reussner, S. Becker, E. Burger, J. Happe, M. Hauck, A. Koziolek, H. Koziolek, K. Krogmann, and M. Kuperberg. The Palladio Component Model. Technical report, KIT, Fakultat fur Informatik, Karlsruhe, 2011.

[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.

[RSCB14] P. Rawat, K. Singh, H. Chaouchi, and J. Bonnin. Wireless sensor networks: a survey on recent developments and potential synergies. *The Journal of Supercomputing*, 68(1):1–48, 2014.

[She10] H. Shen. Content-based publish/subscribe systems. In X. Shen, H. Yu, J. Buford, and M. Akon, editors, *Handbook of Peer-to-Peer Networking*, pages 1333–1366. Springer US, 2010.

[Stu15] O. Stumpf. Security and Trust in the DEECo Component Model. Master's thesis, Department of Distributed and Dependable Systems, Charles Univeristy in Prague, Prague, Czech Republic, 2015.

[SV08] D. Stevanovic and N. Vlajic. Performance of IEEE 802.15.4 in wireless sensor networks with a mobile sink implementing various mobility strategies. In *Local Computer Networks, 2008. LCN 2008. 33rd IEEE Conference on*, pages 680–688, Oct 2008.

[TW11] A. Tanenbaum and D. Wetherall. *Computer Networks*. Pearson Prentice Hall, 2011.

[VH15] E. Vassev and M. Hinchey. Knowledge representation for adaptive and self-aware systems. In M. Wirsing, M. Holzl, N. Koch, and P. Mayer, editors, *Software Engineering for Collective Autonomic Systems*, volume 8998 of *Lecture Notes in Computer Science*, pages 221–247. Springer International Publishing, 2015.

[WSG+13] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. Goschka. On patterns for decentralized control in self-adaptive systems. In R. de Lemos, H. Giese, H. Muller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 76–107. Springer Berlin Heidelberg, 2013.

Web References

[1] WhatIs.com
http://whatis.com

[2] OSGi Alliance. OSGi service platform, core specification, release 5.
http://www.osgi.org/Specifications/HomePage

[3] Eclipse Equinox.
http://www.eclipse.org/equinox

[4] Apache Felix.
http://felix.apache.org

[5] Apache CXF. DOSGi subproject, user guide.
http://cxf.apache.org/distributed-osgi.html

[6] Kevoree.
http://kevoree.org

[7] ASCENS: Autonomic Service-Component Ensembles.
www.ascens-ist.eu

[8] jRESP: Java Runtime Environment for SCEL Programs.
http://jresp.sourceforge.net

[9] ASCENS project case studies.
http://www.ascens-ist.eu/casestudies

[10] Message-Passing.
http://en.wikipedia.org/wiki/Message_passing

[11] OSI model
http://en.wikipedia.org/wiki/OSI_model

[12] Java Messaging Service
http://docs.oracle.com/javaee/5/tutorial/doc/bncdq.html

[13] ActiveMQ project
http://activemq.apache.org

[14] LinkedIn
http://linkedin.com

[15] Kafka project
http://kafka.apache.org

[16] Tuple Spaces
http://c2.com/cgi/wiki?TupleSpace

[17] JavaSpaces specification
http://river.apache.org/doc/specs/html/js-spec.html

[18] Apache River
http://river.apache.org/

[19] GigaSpaces
http://www.gigaspaces.com

[20] SQLSpaces.
http://projects.collide.info/projects/sqlspaces

[21] SemiSpace.
http://www.semispace.org

[22] SimpleIoTSimulator.
http://www.smplsft.com/SimpleIoTSimulator.html

[23] PiccSim.
http://wsn.aalto.fi/en/tools/piccsim

[24] Simulink.
http://mathworks.com/products/simulink

[25] MATLAB.
http://mathworks.com/products/matlab

[26] Modelica.
http://www.modelica.org/

[27] ADEVS.
http://web.ornl.gov/~1qn/adevs

[28] Veins.
http://veins.car2x.org

[29] OMNeT++.
http://omnetpp.org/

[30] SUMO.
http://dlr.de/ts/sumo/en

[31] MiXiM.
http://mixim.sourceforge.net

[32] OpenStreetMap.
http://www.openstreetmap.org

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

[34] jDEECo.
https://github.com/d3scomp/JDEECo

[35] The IEEE 802.15 TG4.
https://ieee802.org/15/pub/TG4.html

[36] ZigBee.
http://www.zigbee.org

[37] Z-Wave.
http://www.z-wave.com

[38] MATSim.
http://www.matsim.org

[39] INET.
https://inet.omnetpp.org

[40] Fraunhofer FOKUS.
http://www.fokus.fraunhofer.de/en

[41] Volkswagen AG.
http://www.volkswagenag.com

[42] DAUM project (in French).
http://daum.gforge.inria.fr