ascens

www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

D1.5: Language Extensions for Implementation-Level Conformance Checking

Grant agreement number: **257414**
Funding Scheme: **FET Proactive**
Project Type: **Integrated Project**
Latest version of Annex I: **Version 2.2 (30.7.2011)**

Lead contractor for deliverable: **CUNI**
Author(s): **Tomáš Bureš (CUNI), Ilias Gerostathopoulos (CUNI), Vojtěch Horký (CUNI), Jaroslav Keznikl (CUNI), Jan Kofroň (CUNI), Michele Loreti (UDF), František Plášil (CUNI)**

Reporting Period: **2**
Period covered: **October 1, 2011 to September 30, 2012**
Submission date: **November 12, 2012**
Revision: **Final**
Classification: **PU**

Project coordinator: **Martin Wirsing (LMU)**
Tel: **+49 89 2180 9154**
Fax: **+49 89 2180 9175**
E-mail: **wirsing@lmu.de**

Partners: **LMU, UNIPI, UDF, Fraunhofer, UJF-Verimag, UNIMORE, ULB, EPFL, VW, Zimory, UL, IMT, Mobsya, CUNI**

SEVENTH FRAMEWORK PROGRAMME

Executive Summary

This document presents two SCEL realizations – jRESP and jDEECo – which implement the concepts of SCEL in Java with the aim to allow for implementation-level conformance checking. jRESP focuses on providing an API for SCEL communication paradigms, thus it allows rapid prototyping of SCEL-based application. On the other hand, jDEECo provides a reification of SCEL geared towards systematic development of large-scale software systems using SCEL concepts. As such jDEECo provides explicit first-class concepts of reusable components and ensembles. Those two frameworks (jRESP and jDEECo) thus act in synergy covering development needs from rapid prototyping to large-scale system development. In addition to describing jRESP and jDEECo as the foundation for implementation-level conformance checking, the document outlines particular directions in the related verification methods. Namely, it presents a method of using JPF for functional verification and demonstrates how performance aspects are represented in the form of knowledge, thus unifying the view on functional correctness and performance aspects of the software.

Contents

1 Introduction

SCEL is a formal modeling language. To be able to reason about implementation-related properties of SCEL-based applications, it is necessary to provide extensions of SCEL in the direction of implementation-level primitives, i.e., to map the concepts of SCEL to the concepts of the implementation level and address implementation-related concepts not explicitly covered by SCEL. In this deliverable we thus overview the basic concepts of SCEL and describe two complementary approaches to allow application development based on SCEL paradigms. The approaches are jRESP and DEECo (in particular the Java-based implementation of DEECo named jDEECo).

jRESP provides faithful mapping of SCEL to Java programming language. Most importantly, jRESP provides an API for SCEL communication paradigms. Being relatively lightweight and prescribing no dedicated architecture or patterns in the target Java application, it is ideal of rapid prototyping of SCEL applications and experimenting with core SCEL concepts.

DEECo, on the other hand, targets systematic engineering of potentially large-scale distributed systems. To do so, DEECo reifies SCEL by adding explicit first-class architectural concepts and restricts SCEL communication paradigms to ease component development and to allow for effective communication in the context of loosely coupled and unreliable communication networks. In particular, DEECo features reusable first-class architectural concepts for both components and ensembles, which allows for explicit system architecture. DEECo itself is thus a component model based on SCEL. The implementation of DEECo in Java is provided by jDEECo, which provides mapping of components and ensembles to Java and provides component runtime that takes care of the component execution and communication.

Having different goals, the two approaches act in synergy and together they address the full range of development needs from rapid prototyping and experiments with SCEL paradigms (addressed by jRESP) to large-scale system development with reusable assets and well-defined architecture (addressed by jDEECo). This positioning and relationship of SCEL, jRESP, DEECo and jDEECo is summarized in Figure 1.

In addition to jRESP and DEECo/jDEECo, the document describes the strategy for verification of functional properties of implementations of SCEL-based applications and also discusses a way of utilizing observed performance of SCEL-based application as the component knowledge.

1.1 Relation to other WPs

By focusing on the implementation aspects of SCEL, the work presented in this report is closely connected to several other work-packages in this project. We briefly overview these connections and synergies below:

- WP3 brings in techniques for knowledge representation and manipulation (KnowLang and KnowLang Reasoner). We plan to integrate these with the approaches to implementation of SCEL-based applications described in this report. In particular, we intend to employ them to define semantics of knowledge both at design and implementation level of SCEL components and ensembles, in particular, this includes handling of derived and uncertain knowledge.

- WP4 focuses on formalization of system requirements (SOTA and GEM). These are planned to be used in the design process of SCEL components and ensembles (as detailed in Section 6.5). Furthermore, WP4 will provide methods for performance monitoring and prediction, for which the implementation aspects of SCEL will serve as a foundation.

- WP5 aims at providing methods for verification of component's code. These methods are to be based on the hereby proposed mapping of SCEL-based paradigms to the Java-language.

- WP6 integrates tools developed within the project to a common workbench. The approaches to executing SCEL-based applications (i.e., jRESP and jDEECo) are thus included in WP6 as component runtime platforms.

- WP7 focuses on case-studies. These are going to be based on jRESP and jDEECo. In particular, the initial results of using jDEECofor addressing the e-mobility case-study are already part of the D7.2 [S$^+$12].

- WP8 aims at engineering of service components and related best practices. In this respect, this report describes initial proposal for high-level design of SCEL-based applications. Additionally, the components implemented using jRESP and jDEECo concepts are going to be included in the service component repository, which is part of D8.2 [HBGK12].

1.2 Structure of the Report

The structure of the deliverable is as follows: In Sect. 2, the SCEL design concepts are described. In Sect. 3 jRESP is described. DEECo and its Java-based implementation jDEECo is described in Sect. 4 and Sect. 5 respectively. Sect. 5 further provides discussion of verification of functional properties of SCEL-based application implemented in jDEECo. Sect. 6 reports on initial experiments with a design method for distilling SCEL-based architecture (with DEECo flavour) from initial system requirements. Sect. 7 describes work in progress on how performance is viewed as a form of knowledge and outlines the methods for collecting and interpreting performance indicators. Sect. 8 concludes the deliverable and states the ongoing steps.

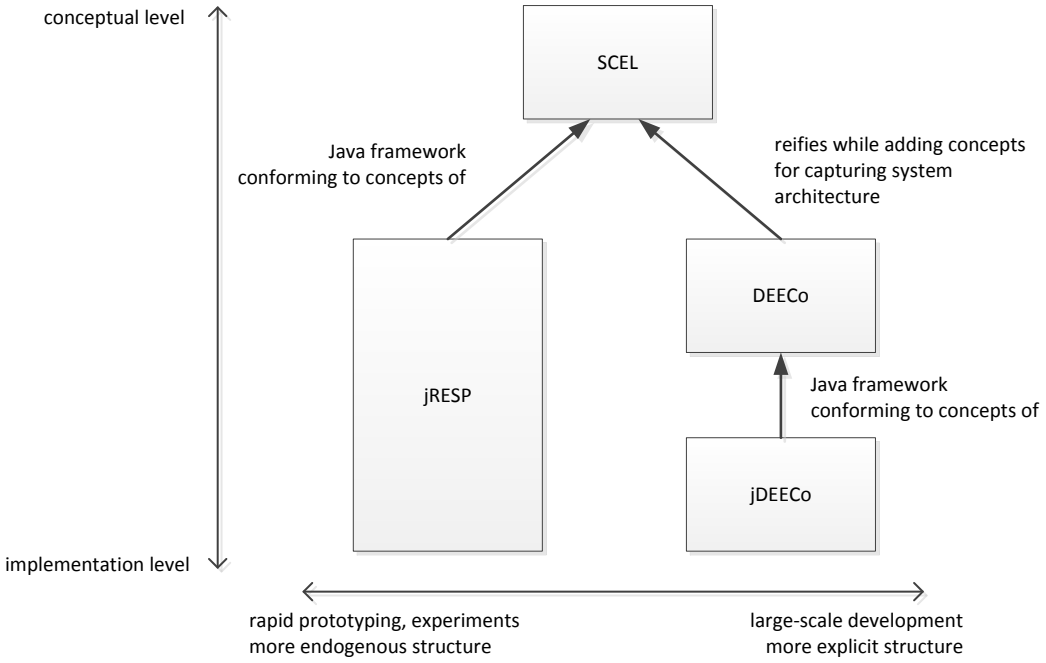

Figure 1: Relationships among SCEL, jRESP, DEECo, and jDEECo

2 SCEL: Design Principles

SCEL provides abstractions explicitly supporting autonomic computing systems in terms of *Behaviors*, *Knowledge* and *Aggregations*, according to specific *Policies*.

Behaviors describe how computations progress. These abstractions are modeled as processes executing actions, in the style of standard process calculi. *Interactions* come in when components access data in the knowledge repositories of other components. *Adaptation* emerges as the result of knowledge acquisition and manipulation.

Knowledge provides the high level primitives to manage pieces of relevant information coming from different sources. Knowledge is represented through items stored in repositories. Knowledge items contain either *application data* or *awareness data*. The former are used to determine the progress of component computations, while the latter provide information about the environment in which the different components are running (e.g. monitored data from sensors) or about the actual status of an autonomic component (e.g. about its current position or the remaining battery's charge level). We assume that each knowledge repository handling mechanism provides three abstract operations that can be used by autonomic components to *add* new knowledge to the repository, to *retrieve* knowledge from the repository and to *withdraw* knowledge from it.

Aggregations describe how different entities are brought together to form *components* and *systems*, and to offer the possibility to construct the *software architecture* of autonomic systems. The composition of components and their interactions is implemented by exploiting the notion of *interface* that can be queried to determine the attributes and the functionalities provided and/or required by components. *Ensembles* are specific aggregations of components that represent *social or technical networks* of autonomic components. The key point is that the formation rule is dynamic, based on the knowledge endogenous to components: components of an ensemble are connected by the interdependency relations established in their interfaces. Therefore, an ensemble is not a rigid fixed network, but rather a dynamic graph-like structure where component links are dynamically established.

Policies control and adapt the actions of the different components, in order to guarantee the achievement of specific goals, or the satisfaction of specific properties. Since few assumptions can be made about the operational environment, that is frequently open, highly dynamic, and possibly hostile, the ability of programming and enforcing a finer control on behavior is essential to assure that, for instance, valuable information is not lost. Policies are the mean to guarantee such control. Interaction policies and Service Level Agreement (SLA) policies provide two standard examples of policy abstractions. Other examples are security properties maintaining the right linkage between data values and their associated usage policies (data-leakage policies), or limiting the flow of sensitive information to untrusted sources (access control and reputation policies).

All these abstractions are aggregated by means of the notion of autonomic component. An *autonomic component* $\mathcal{I}[\mathcal{K}, \Pi, P]$ consists of:

1. an *interface* $\mathcal{I}$, publishing and making available structural and behavioral information about the component itself;

2. a *knowledge manager* $\mathcal{K}$, managing both application data and awareness data, together with the specific handling mechanism;

3. a set of *policies* Π, regulating the interaction between the different internal parts of the component and the interaction of the component with the others;

4. a *process* P together with a set of process definitions that can be dynamically activated. Some of the processes in P perform local computation, while others may coordinate processes inter-

action with the knowledge repository, and deal with the issues related to adaptation and reconfiguration.

A component's interface can be inquired to extract information about the component, its status or its execution environment, as well as the services offered by the component. In fact, the interface provides a set of *attributes* characterizing the component itself. Among these attributes, attribute id is mandatory, and is bound to the name of the component. Additional attributes might, for instance, indicate the battery's charge level and the component's GPS position. Suitable attributes are also used to indicate the provided services and their signature. Notably, the whole information provided by the component interface is stored in the local knowledge of the component and therefore can be dynamically manipulated by means of the operations provided by the knowledge repositories' handling mechanisms.

SCEL puts together concepts and abstractions for development of autonomic computing systems. It specifies them, however, at a relatively high level. To be able to evaluate properties (including non-functional ones) of real applications, it is necessary first to map the SCEL concepts to those on the implementation level, and second to provide a sufficient set of SCEL extensions to address platform-specific issues not covered by SCEL itself. In the project, we face this via two complementary approaches – jRESP and jDEECo, which are described in the following sections.

3 jRESP – A Runtime Environment for SCEL Programs

In this section we present jRESP[1]. This is a Java runtime environment that aims at providing programmers with a framework that allows for the development of autonomic and adaptive systems according to the SCEL [DFLP11, DFLP12] paradigm. The main objective of jRESP is to be a realization of SCEL in Java, which is faithful enough and easy to start with to be suitable for rapid prototyping and experiments with SCEL.

3.1 Design Principles

SCEL identifies linguistic constructs to model in a uniform way the control of computation, the interaction among possibly heterogeneous components, and the architecture of systems and ensembles. A SCEL *program* consists of a set of (possibly heterogeneous), components, equipped with their own knowledge repository, that concur and cooperate to achieve a set of *goals*. The underlying communication infrastructure is not fixed, but can change dynamically during the computation. That said, a first principle followed in design and implementation of jRESP is avoiding *centralized control*. In particular, components are able to interact with each other by simply relying on the available communication media.

Moreover, in the definition of SCEL, some categories, like *knowledge* and *policy*, are not fixed but can be identified time to time according to the specific application domain or to the taste of the language user. Other mechanisms, such as, for instance, the underlying communication infrastructure, are not considered at all, and are, instead, *abstracted* in the operational semantics. For these reasons, the while framework is parametric with respect to specific implementations of above mentioned features. Design patterns have been largely used in jRESP to simplify development of specific implementations of knowledge, policies and underlying communication infrastructure.

Finally, to simplify the integration with other tools/framework (such as ARGoS [PTO+11] and DEECo[2]), jRESP relies on *open technologies* like, for instance, json [3]. Such tools, by providing

[1] http://code.google.com/p/jresp/

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

[3] http://www.json.org

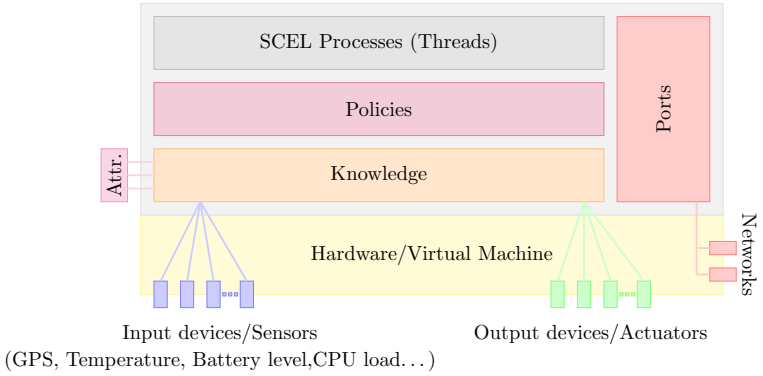

Figure 2: Node architecture

mechanisms for data-interchange format, simplify the interactions between heterogeneous network components and provide the basis on which different runtime for SCEL programs can cooperate.

3.2 Components

The central element of jRESP is the class Node. This class provides the implementation for a generic SCEL component[4]. The overall infrastructure of a generic node is reported in Figure 2.

We assume that each node is executed over a virtual machine or a physical device that provides the access to input/output devices and to network connections. Each node contains: a *knowledge*; a set of running processes/threads; and a *policy*. Structural and behavioral information about a node can be collected into an *interface* via a set of *attribute collectors*. Nodes interact with each other via *ports*. These provide mechanism for supporting both *one-to-one* and *group* communications.

3.3 Knowledge

In jRESP a Knowledge is an interface that identifies a generic *knowledge repository* and indicates the high level primitives to manage pieces of relevant information coming from different sources. This interface contains the methods that can be used to add, get, and query from piece of knowledge from a repository.

Class Tuple, defined in the same package as Knowledge, identifies the basic information item. It consists of a sequence of values, (i.e. Objects), that can be collected into a knowledge repository.

To identify the tuples to get/query from a knowledge repository, an instance of class Template is used. A Template consists of a sequence of TemplateField. The latter is an interface providing the single method match(Object o): boolean. Class Template relies on method match to verify if a Tuple can be selected or not. Indeed, a tuple matches a template if both have the same number of elements and the corresponding element matches.

Currently, a single implementation of interface Knowledge is available in jRESP. Class TupleSpace in package org.cmg.ml.resp.knowledge.ts provides an implementation for a Linda [Gel85] tuple space.

[4]From now on we will use *node* to refer to instances of class Node, while *component* will indicate a SCEL component.

3.4 Sensors

To retrieve data from external input devices, nodes are equipped with *sensors*. These are instances of class Sensor that can be used to identify a generic source of information. Each sensor can be associated to a logical or physical device providing data that can be used by processes and that can be the subject of adaptation. Each sensor exports data as a *tuple* that is made available in the node knowledge.

3.5 Actuators

Instances of class Actuator can be used to send data to external components or devices. Similarly to Sensor, an instance of class Actuator can be used to *control* an *external* component that identifies a logical/physical actuator. Processes can pass values to actuators by relying on standard SCEL operations on knowledge.

3.6 Attributes and Atribute Collectors

An attribute is defined as a pair *(name,value)*. Attributes can be published in a node interface via *attribute collectors*. Attribute collectors can be implemented by extending abstract class AttributeCollector. When a node receives a request for an attribute *a*, the corresponding collector is selected. Hence, this interacts with the node knowledge to compute the actual attribute value.

3.7 Ports and Network Infrastructure

Each node is equipped with a set of *ports* that are able to handle both *point-to-point* interactions and *group* interactions (*ensemble* oriented). Indeed, a port provides a generic communication channel that follow a specific communication protocol.

Currently the following ports are available:

- InetPort, this kind of ports uses TCP to *point-to-point* interactions and UDP for the *group* ones;
- ServerPort, in this case a centralized server is used to collect and dispatch nodes' actions;
- VirtualPort, this is used to *simulate* nodes running on *virtual* devices.

Each port is identified by a physical address. For instance, in the case of InetPort, this is the *inet-address* associated to the socket where a thread is waiting for incoming connections.

3.8 Agents

Agents are the jRESP active computational units and are threads used to program the behavior of SCEL processes. The abstract class Agent provides the methods to manage shared knowledge repositories by withdrawing/retrieving/adding information items from/to a knowledge repository. These methods extend the one considered in Knowledge with another parameter identifying the, (possibly remote) node where the target knowledge repository is located.

A target can be either a *point-to-point* address, a *group* or *self*. These are implemented via classes PointToPoint, Group and Self, respectively. A point-to-point address univocally identifies the target of the considered action. A *group* identifies all the nodes that satisfy a given predicate on nodes attributes. Special target Self is used to refer to the node where an agent is running.

To program a specific process behavior, sub-classes of Agent must provide an implementation for abstract methods doRun(). The latter is the one that is invoked when an agent is *executed* at a node.

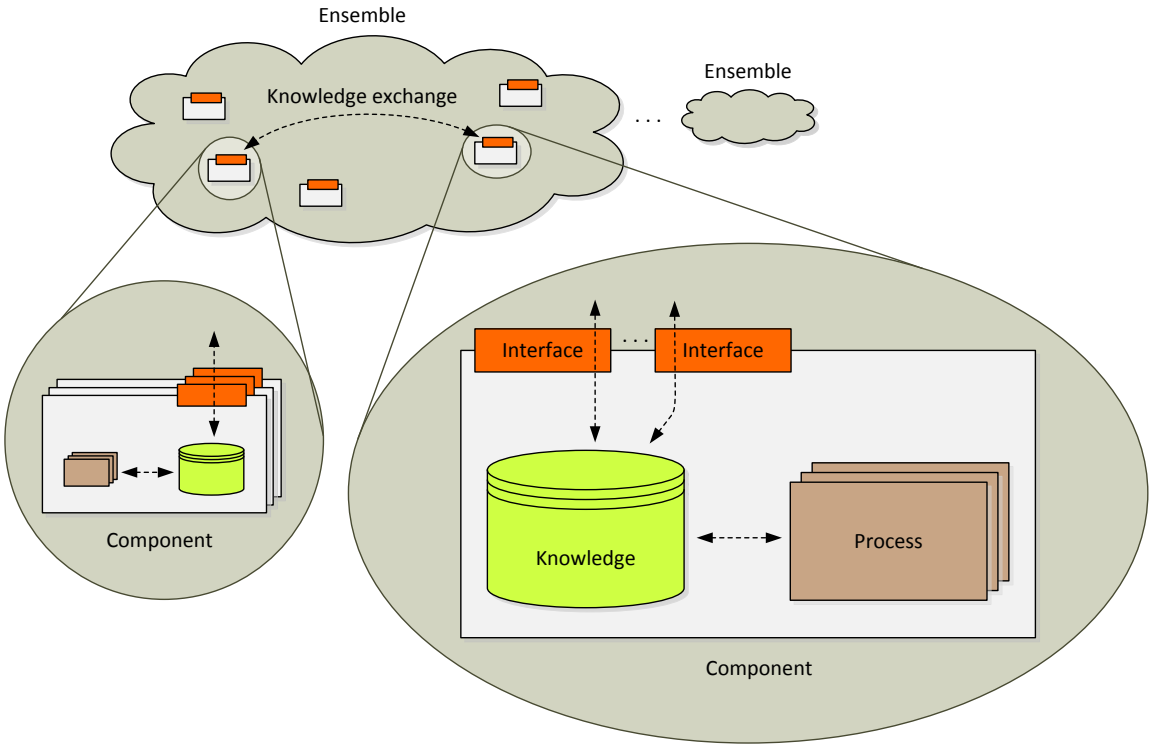

Figure 3: Basic DEECo entities and their relationship

3.9 Policies

Policies are attached to a node in order to control and adapt the performed actions, to guarantee the achievement of specific goals, or the satisfaction of specific properties. In particular, when an agent invokes a method, this is first delegated to the policy associated.

Policies are organized in a *stack*. The policy at one level relies on the one at the level below to actually execute SCEL actions. The policy at the lower level is the one that allows any operation.

4 DEECo – Dependable Emergent Ensembles of Components

In this section, we describe the DEECo component model [KBPK12], which is a reification of SCEL targeting large-scale system development with explicit architectural concepts of components and ensembles.

4.1 Design Principles

Being a specialization of SCEL, DEECo is based on same main design concepts (Figure 3). Specifically, DEECo allows for the design of systems consisting of autonomous, self-aware, and adaptable *components*, which are implicitly organized in groups called *ensembles*.

Reifying the SCEL concepts, the main idea is to manage all the dynamic aspects of the service-component environment by externalizing the distributed communication between service components into a component framework. We propose a particular way of perceiving a component, i.e., as a self-aware unit of computation, relying solely on its *local* knowledge that is subject to external modification during the execution time. The communication of such components is represented as *knowl-*

edge exchange, entirely externalized and automated within the DEECo runtime framework. Thus, the components are programmed as autonomous units, without relying on whether/when the knowledge exchange is performed. This makes the components very robust and suitable for rapidly-changing environments.

Conceptually, the role of the DEECo runtime framework is to perform scheduling of components' processes, and carry out knowledge exchange of ensembles.

These DEECo concepts can be easily mapped back to the SCEL concepts. We will show such mapping for the particular DEECo concepts in the following sections.

4.2 Component Structure

Similar to SCEL, a component is an autonomous unit of deployment and computation; it consists of *knowledge*, exposed via a set of component *interfaces*, and *processes* (Figure 3). However, unlike SCEL, DEECo components do not comprise specific *policies*. Instead, DEECo prescribes the same policy for all components; the semantic role of the policy in knowledge access and exchange will be explained in Section 4.2 and Section 4.3, respectively.

4.2.1 Knowledge

Knowledge reflects the state (i.e., data) and functionality (i.e., executable functions) of the component. As a SCEL specialization, DEECo provides a general description of both the *knowledge representation* and *knowledge handling* mechanisms.

Knowledge representation is defined as a mapping from knowledge *identifiers* to knowledge *values*. The knowledge repository is a bag of pairs of the form $\langle \text{“}identifier\text{”}, value \rangle$ – the *knowledge items*. The knowledge values are either executable functions without side effects or statically-typed (structured) data. In particular, the identifier of a sub-value of a structured value is represented as a structured name reflecting the value's internal structure. The top-level value is always the whole component knowledge, identified by the component's id. For example, if the component with id $Car1846a$ has a knowledge item named $position$ consisting of two floating-point-number fields named x and y, the identifier of the x field will be $Car1846a.position.x$.

In the knowledge representation, only the unstructured values (e.g., $Car1846a.position.x$) are actually stored, while the structured ones (e.g., $Car1846a.position$) are regarded as an aggregation of their sub-values. For example, in case of the $Car1846a$ component, its knowledge representation would contain only the tuples $\langle \text{“}Car1846a.position.x\text{”}, v_x \rangle$ and $\langle \text{“}Car1846a.position.y\text{”}, v_y \rangle$, for some concrete values v_x and v_y. The value of $Car1846a.position$ would be aggregated from these tuples.

In addition to user-defined value types, DEECo introduces two collection types – map and $list$. While map is represented in the same way as described above (i.e., the keys of the values in the map are directly used in the structured value identifiers), the values in a $list$ are given implicit identifiers based on their position in the list. For example, the individual values of the list $Car1846a.checkpoints$ have the identifiers $Car1846a.checkpoints.0$, $Car1846a.checkpoints.1 \ldots Car1846a.checkpoints.(n-1)$, where n is the size of the list.

Knowledge handling mechanism in DEECo mediates reading/writing/withdrawing the above-described knowledge values. Specifically, focus is put on structured values. These are in DEECo manipulated by the component's processes as in-memory structures/objects. Therefore, as a structured value is serialized into a set of tuples containing the value's unstructured sub-values,

```
component Car1846a ... :
    knowledge:
        id: ComponentID = "Car1846a";
        position: Position = {
            x: Real = 1.0;
            y: Real = 2.0;
        };
        checkpoints: list Position = { {x = 1.5; y = 1.2;}; ... };
        findNearest: fun(in checkpoints: list Position, out nearest: Position) = { ... };
        ...
    ...
```

Figure 4: DEECo DSL knowledge representation example

the knowledge representation ensures aggregation/splitting of the structured values when reading/writing. This, in addition to the identifier convention for the collection types (i.e., map and list), represents a general mechanism to access structured values. All the knowledge handling operations, including reading/writing structured values and collections, are performed atomically.

Since DEECo employs an implicit interaction mechanism among components, the knowledge handling mechanism limits a component to access only its own knowledge. Therefore, from the SCEL perspective, only the target $self$ is allowed by the DEECo component policy to all knowledge handling actions, i.e., get/qry/put (the inter-component knowledge access is discussed in Section 4.3).

Building on our close collaboration with WP3, we expect the knowledge representation and handling semantics to be further elaborated towards KnowLang (D3.2, [VHM+12]). A promising approach appears to be extending the basic DEECo knowledge value types with the KnowLang concepts for representing factual knowledge, in terms of concept/object trees and uncertain knowledge. In the future, we intend to employ the product of the KnowLang Compiler as the actual representation format of component knowledge. This way, there is a potential to extend the DEECo knowledge handling mechanism with the features of the KnowLang Reasoner (i.e., the ASK and TELL operators), as outlined in Section 5.3, thus equip the components with derived knowledge.

Before employing KnowLang as envisioned above, we use a simple DSL (Domain-Specific Language) for describing the knowledge structure of a component, which integrates well with the DEECo DSL capturing the other DEECo concepts. In the DSL, the knowledge values are represented as hierarchically organized key/value sets, with specific syntactic constructs for capturing lists, executable functions, and function parameters. An example of knowledge representation is shown in Figure 4. Here, the knowledge representation of the component $Car1846a$ comprises a structured value position of type Position (consisting of two unstructured values x and y), a list of Position values named checkpoints, and a function findNearest with one input parameter checkpoints (of type list of Position) and one output parameter nearest (of type Position).

4.2.2 Interface

In general, the knowledge of a component is exposed to the environment via a set of interfaces. An interface represents a partial view on a component's knowledge. Specifically, interfaces of a single

```
interface Car:
    position: Position;
    checkpoints: list Position;

component Car1846a features Car, ... :
    ...
```

Figure 5: DEECo DSL interface representation example

component can overlap, and multiple components can provide the same interface, thus allowing for polymorphism. Technically, an interface relates to a selection of identifiers and types of knowledge values, forming a "template" for component knowledge.

In SCEL, all the DEECo interfaces would be merged into a single SCEL component interface.

An example of an interface of the $Car1846a$ component is shown in Figure 5, exposing the position and checkpoints knowledge values.

4.2.3 Process

Each process of a component is essentially a thread, which operates upon the knowledge of the component. Compared to SCEL, DEECo distinguishes the notions of *process* – the activity performing computation – and *function* – the definition of a computation.

Specifically, a process employs a function from the knowledge of the component to perform a computation. Since all functions are assumed to have no side effects, a process comprises a mapping of the component's knowledge to the actual parameters of the employed function (*input knowledge*), as well as a mapping of the return value back to the knowledge (*output knowledge*). In other words, the responsibility of a process is to read the input knowledge, call the associated function, and write the outcome of the function back into the knowledge.

Being active entities of computation, DEECo processes are subjects to scheduling. A process can be either *periodic* or *triggered*. Since the processes run concurrently, it is necessary to apply suitable knowledge-access policies to maintain knowledge consistency. Reflecting the anticipated use of the two process kinds, DEECo applies a tailored policy to either of them.

In SCEL, both DEECo processes and functions would be mapped to specific kinds of SCEL processes.

Periodic process is executed repeatedly in a given period. The knowledge-access policy for a periodic process ensures that reading the input knowledge, as well as writing the output knowledge, is considered an atomic step. Nevertheless, interleaving of other processes is possible between the reading and writing steps. Thus, such interleaving may lead to the execution of a process with "old" input values. Such periodic processes are suitable for repetitive tasks, such as processing sensor data, or performing continuous-time control.

Triggered process is executed asynchronously, whenever (a part of) its input knowledge changes, or whenever a given condition on the component's knowledge – the process's *guard* – is satisfied. The knowledge-access policy for a triggered process ensures that all three steps of the process – reading the input knowledge, computing the outcome of the employed function, and writing the output knowledge are (together) considered to be a single atomic step. This way, no interleaving of other processes is allowed between the reading and writing steps. This is very important for

```
component Car1846a features Car, ... :
    knowledge:
        position: Position = ...;
        checkpoints: list Position = { ... };
        nextCheckpoint: Position = ...;
        ...
        getPosition: fun(out position: Position) = { ... };
        findNearest: fun(inout checkpoints: list Position, out nearest: Position) = { ... };
        ...
    process positionMonitor:
        function: getPosition
        input: { }
        output: { position }
        scheduling: periodic( 100ms )
    process navigate:
        function: findNearest
        input: { checkpoints }
        output: { nextCheckpoint, checkpoints }
        scheduling: triggered( position == nextCheckpoint )
    ...
```

Figure 6: Example of a DEECo DSL process specification

triggered processes, since the (potential) knowledge inconsistencies caused by interleaving may be unrecoverable (e.g., in case the triggering condition is not satisfied anymore). Such triggered processes are suitable to handle asynchronous events, such as service requests.

Precise semantics of process execution naturally depends on the way the atomicity is implemented, which is still an open question. To this date, we use an implementation based on locking and an implementation based on transactions.

In general, the DSL features syntactic constructs for capturing component processes; an example is shown in Figure 6. Here, the periodic process positionMonitor employs the getPosition function to repeatedly check a sensor for the current position and stores it as the position knowledge value. Similarly, the triggered process navigate employs the findNearest function to compute the nextCheckpoint from the list of checkpoints, while trimming the list; the process is triggered then the car reaches the current checkpoint (i.e., position equals nextCheckpoint). As for knowledge consistency, since navigate is a triggered process, the consistency policy ensures that when written the values of nextCheckpoint and checkpoints correspond to the outcome of findNearest applied on the original value of checkpoints (i.e., checkpoints was not changed while findNearest was executing).

4.3 Component Composition and Interaction

Similar to SCEL, component composition in DEECo is flat, captured implicitly via a dynamic involvement in an *ensemble* (Figure 7). An ensemble comprises a single *coordinator* component and multiple *member* components; the set of components forming an ensemble is, for the purpose of specification, referred to also as a set of coordinator-member pairs sharing the same coordinator. However, in contrast to SCEL, a DEECo ensemble is a first-class concept.

An involvement of a component in an ensemble in either of the roles coordinator/member is determined dynamically by the runtime framework, according to the *membership condition* of the en-

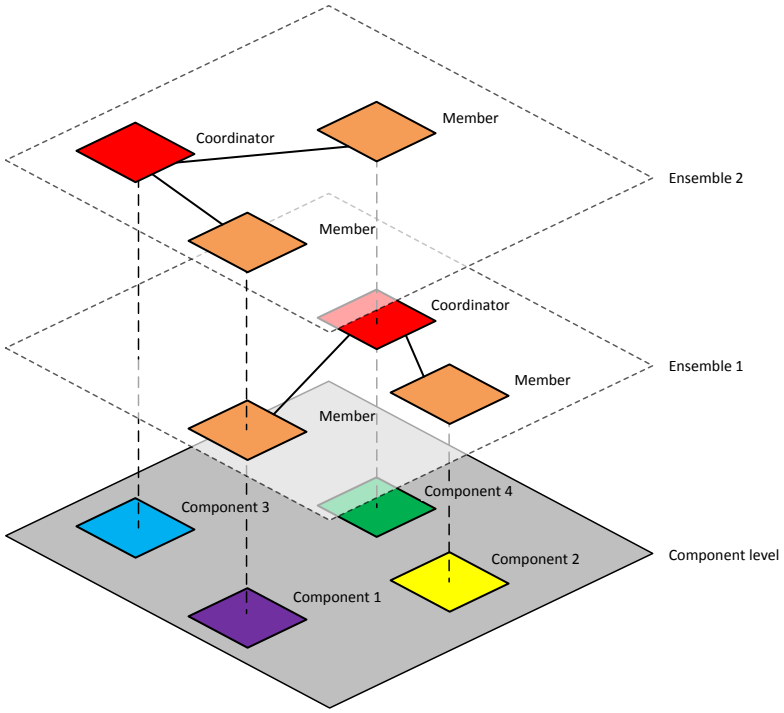

Figure 7: Composition of components into ensembles in DEECo

semble. A membership condition constitutes definition of the required *interface* (Section 4.2.2) of the coordinator and of the member components (i.e., the knowledge that is necessary for a component in order to take part in an ensemble, either as coordinator or member) and the *membership predicate* (i.e., the condition over these interfaces, under which the components featuring these interfaces represent the coordinator-member pair of the ensemble).

The interaction among the components forming an ensemble takes the form of *knowledge exchange*. In contrast to SCEL, the interaction is implicit, i.e., a component does not proactively access the knowledge of the other components. Instead, knowledge access and exchange is performed by the runtime framework. The only allowed form of interaction among components is, similar to SCEL, interaction between a member and the coordinator of an ensemble. This allows the coordinator to apply various communication policies.

Here, an important idea is that the components do not perceive their membership in an ensemble. They operate only upon their own local knowledge, which gets implicitly updated by the runtime framework (via knowledge exchange) whenever the component is part of an ensemble.

In SCEL, a DEECo ensemble is to be represented by a component, designated to evaluating the membership condition and performing the knowledge exchange.

4.3.1 Membership

A membership predicate declaratively expresses the condition under which two components represent a pair coordinator-member of the associated ensemble. The predicate is defined upon the interfaces of the components. The membership predicate is evaluated by the runtime framework when necessary.

In general, the same component can be at the same time the coordinator and a member of a single ensemble. Also, a single component can be member/coordinator of multiple ensembles (Figure 7).

```
interface TrafficAwareCar:
    position: Position;
    otherCars: list Position;
    ...

ensemble CarPositionExchange:
    coordinator: TrafficAwareCar
    member: TrafficAwareCar
    membership:
      distance(coordinator.position, member.position) ≤ THRESHOLD
    knowledge_exchange:
        coordinator.otherCars ← members.reduce(position)
    scheduling: periodic( 100ms )
```

Figure 8: Example of a DEECo DSL ensemble specification

In the latter case, i.e., when a component satisfies the membership predicate of multiple ensembles, we envision a mechanism to decide whether all or only a subset of the potential coordinator-member pairs should be formed. Currently, we employ a simple mechanism of a partial order over ensembles for this purpose (the "maximal" ensemble of the comparable ones is formed, the ensembles which are incomparable are formed simultaneously). For example, providing a static priority in ensemble definition yields such a partial order.

Figure 8 shows an example of a membership predicate, according to which two components having the TrafficAwareCar interface form the coordinator-member pair if the member's distance from the coordinator is less or equal to a fixed THRESHOLD. Note, that the predicate is symmetric, meaning that whenever two components A and B form a coordinator/member pair, then also B and A form a coordinator/member pair. Thus, both A and B take the role of both the coordinator and member, each in a different ensemble (identified by its coordinator).

4.3.2 Knowledge Exchange

The knowledge exchange embodies the interaction between the coordinator and all the members of an ensemble. Being implicit, knowledge exchange is carried out by the runtime framework. Thus, it is up to the runtime framework when/how often knowledge exchange is performed. Similarly to component processes (Section 4.2.3), knowledge exchange can be carried out either periodically or when triggered. In the former case, the knowledge consistency policy is similar to a periodic process. However, in the latter case, the knowledge consistency policy is more complex than it is for a triggered process, since the triggering condition refers to a coordinator-member pair of components. The whole triggered knowledge exchange (i.e., exchange with all members) is, similarly to a triggered process, executed as a single atomic step.

It is the task of the runtime framework to ensure that membership of the interacting components holds when performing knowledge exchange. Technically, the knowledge mapping is represented by a *mapping function*, which maps the knowledge of the coordinator to the knowledge of all the members (i.e., the set of all members is passed as an argument), and vice versa.

Figure 8 shows an example of a mapping function, which periodically aggregates the position of all members into the otherCars knowledge value of the coordinator.

5 jDEECo – A Runtime Environment for DEECo Applications

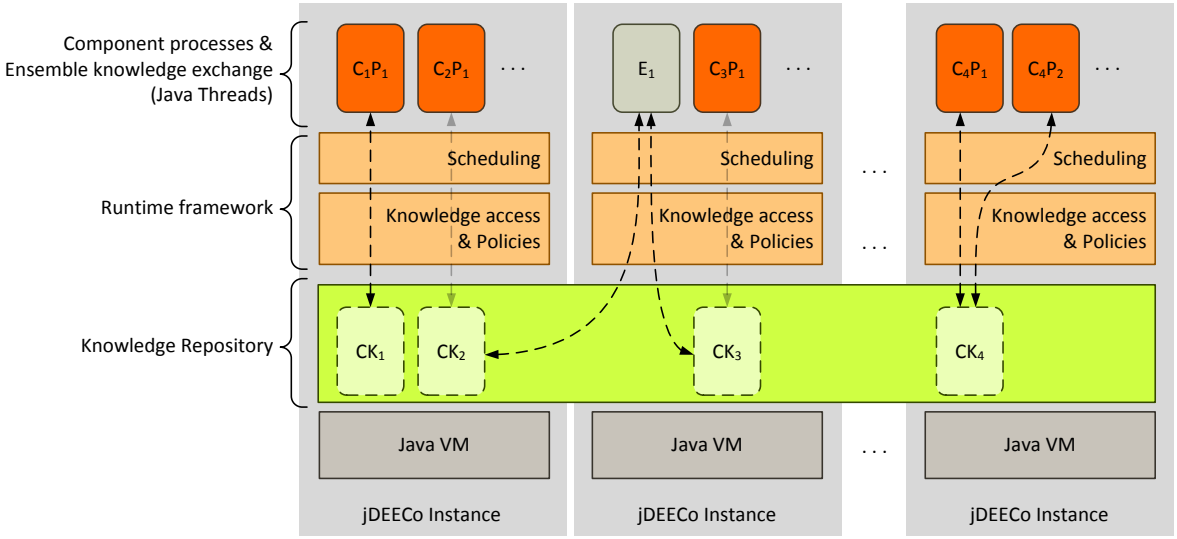

Figure 9: jDEECo architecture

In this section, we describe jDEECo – an implementation of the DEECo component model in Java. jDEECo maps the abstract SCEL concepts, refined by DEECo, to implementation-level primitives. In this section we thus deal with the component and ensemble definition in Java, their in-memory representation, and the threading model. Technical details on jDEECo integration in ASCENS Service Development Environment (SDE) are given in D6.2 [CHK+12]. Additionally, we focus on the concepts relevant to implementation-level verification techniques, described in Section 5.4.

5.1 Component

A component definition has the form of a Java class. The initial knowledge of the component is captured by means of fields and static methods of this class, while the processes are captured by means of specifically-annotated static methods of the class. The initial knowledge can contain both structured (i.e., structured knowledge, map, and list) and primitive knowledge values (i.e., serializeable Java objects). All the DEECo-specific information is captured by means of dedicated class/method/argument annotations.

In contrast to the conceptual description of a component (Section 4.2), the Java implementation of a component does not comprise interfaces. Instead, the set of supported interfaces is implicit, i.e., all interfaces that structurally match the component's knowledge are assumed to be supported by the component (similar to duck typing in dynamic languages).

Figure 10 shows an example of a Java implementation of the $Car1846a$ component from Section 4.2.

5.2 Ensemble

Ensemble definition takes also the form of a Java class. Both the membership predicate and mapping function are captured by means of specifically-annotated static methods of this class. Similar to processes, these methods are annotated with DEECo-specific information.

```
@DEECoComponent
public class Car1846a extends ComponentKnowledge {

    public Position position;
    public List<Position> otherCars;
    ...

    @DEECoInitialize
    public static ComponentKnowledge getInitialKnowledge() {
        /* return initialized instance Car1846a */
    }

    @DEECoProcess
    @DEECoPeriodicScheduling(100)
    public static void positionMonitor(@DEECoOut("position") OutWrapper<Position> position) {
        /* obtain the position from a sensor */
    }
    ...
}
```

Figure 10: Example of a DEECo component implementation in Java

In contrast to the conceptual description of ensemble (Section 4.3), Java implementation of an ensemble does not comprise the definition of the member and coordinator interfaces. Instead, these interfaces are defined implicitly as a union of the knowledge values serving as in/inout arguments of the membership, and mapping functions. Technically, the member and coordinator arguments are distinguished by identifier prefixes.

Figure 11 shows an example of a Java implementation of the ensemble from Section 4.3.

5.3 Runtime Framework

The jDEECo runtime framework implementation provides all the functionality necessary to schedule component processes and perform knowledge exchange (Figure 9).

Concerning component knowledge (e.g., CK_1 – knowledge of component C_1 – in Figure 9), it is stored in a knowledge repository, where it can be accessed by component processes and knowledge exchange of ensembles (while obeying knowledge access policies). In general, this repository may be distributed and thus shared among several jDEECo framework instances. As a part of the knowledge handling mechanism of the knowledge repository, KnowLang Reasoner (D3.2, [VHM+12]) should be used to implement operations over the derived and uncertain knowledge. For the purpose of verification (Section 5.4), jDEECo employs a specialized implementation of the knowledge repository based on a local hash map, implementing knowledge consistency policies via locking.

Concerning DEECo component processes (e.g., C_1P_1 – process P_1 of component C_1 – in Figure 9), they are being executed as regular Java threads. Nevertheless, their access to the knowledge repository is managed according to the knowledge consistency policies. Specifically, threads executing triggered processes are blocked till their triggering condition holds true, while threads executing periodic processes are between two subsequent runs blocked for the duration of their period.

Concerning knowledge exchange of ensembles (e.g., E_1 in Figure 9), the scheduling of the associated mapping functions is similar to component processes. However, the membership function is evaluated before each run of the mapping function, so that the knowledge exchange is applied only to

```
@DEECoEnsemble
@DEECoPeriodicScheduling(100)
public class CarPositionExchange {

    public static final int THRESHOLD = 20;

    @DEECoEnsembleMembership
    public static boolean membership(
            @DEECoIn("member.position") Position mPosition,
            @DEECoIn("coordinator.position") Position cPosition) {
        return distance(cPosition, mPosition) <= THRESHOLD;
    }

    @DEECoEnsembleMapper
    public static void map(
            @DEECoInOut("members.position") OutWrapper<Position[]> positions,
            @DEECoInOut("coordinator.otherCars") OutWrapper<List<Position>> otherCars) {
        otherCars.set(positions.get());
    }
}
```

Figure 11: Example of a DEECo ensemble implementation in Java

valid coordinator-member pairs of components.

5.4 Techniques for Verification of Component and Ensemble Properties at Implementation Level

For verification of SCEL-based applications at the implementation level, we use Java PathFinder (JPF) [JPF]. Specifically, we applied the JPF on jDEECo with the local knowledge repository. Currently, we support verification of the properties upon the knowledge, encoded via asserts and exception mechanism. The spectrum of verified properties will be extended to support a wider range of properties in the future.

Since JPF is, in principle, a special virtual machine, it must reflect the language changes coming with new versions. In addition, it comes with its own, limited, implementation of standard libraries. For this reason, it may happen that not all language constructs are supported by the VM, and that programs behave differently on JPF than, for instance, on the Oracle Java VM. To be able to successfully verify properties of an application, it is often necessary to modify the code to comply with the set of supported language constructs and libraries. In this project year, we have studied what is supported by JPF and modified the jDEECo implementation to be able to model-check it through JPF. Moreover, we have optimized the jDEECo implementation to make verification reasonably fast. This is a necessary prerequisite for verification of larger systems built upon jDEECo.

6 On-going work on High-level Design of SCEL-based Applications

In this section, we describe the work in progress on a high-level design method to build SCEL-based systems. The results of this work belong to the development methods (D8.3) planned for year 3, nevertheless we have started some initial research and experiments in this field, which help assess that

the implementation concepts proposed in the document are feasible and appropriate from the high-level design. In particular, this has to be further iteratively updated with respect to progress on the work on SOTA and GEM (targeting formalization of requirements). As further shown in the section, we have performed such initial experiments on a small case study from the cloud environment. In the same vein, we are currently working in cooperation with VW on a design experiment that employs the e-mobility case study – initial results are reported in D7.2 [S$^+$12].

The eventual goal is to provide a well-structured design technique for all the phases of the software development process, from early requirements to an implementation based on concepts of components and ensembles with the refined semantics as proposed by the DEECo component model, which also supports mapping to general SCEL components and ensembles (D1.2, [PBN$^+$12]).

Overall, the design method envisions employment of the methods for describing and formalizing requirements (SOTA, GEM and Poem) as discussed in Section 6.5, and on methods to representing knowledge and inferring derived and uncertain knowledge (KnowLang and KnowLang reasoner) as discussed in Section 6.6.

6.1 Overview

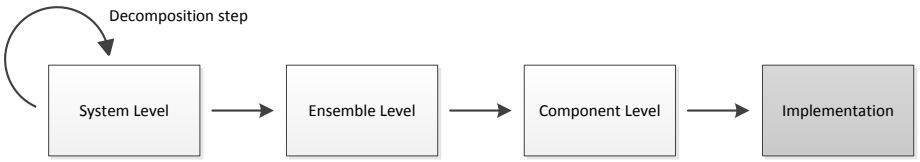

Figure 12: High-level system design overview.

Following the idea of the top-down design paradigm, the design process is based on systematic decomposition and refinement of the system specification (Figure 12). It consists of: *system level* design, *ensemble level* design, and *component level* design, followed directly by *implementation*.

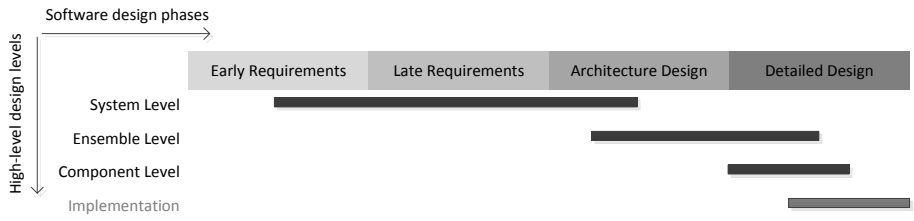

Figure 13: Span of the DEECo high-level system design throughout the design process.

The three levels cover all the phases of the software design process starting from the *early requirements* phase, followed by *late requirements* and *architecture design* phases, ending up with the *detailed design* phase (Figure 13). Although similarities to goal-based methodologies like Tropos

[BGG+03] can be found, the DEECo design process focuses on an integrated view on the system's requirements and architecture.

In the rest of the section we describe the three levels of the design process. For each level, the main concepts, together with their (graphical and/or textual) representation, and relevant examples are presented.

6.2 System Level

As a starting point of the design process, similar to goal-oriented requirements engineering [vL01], it is important to answer the questions "which (global) goals must be achieved?" and "which (system) attributes must be maintained?". The next question is "who is responsible for achieving/maintaining these attributes?". In our method, these questions must be answered so as to obtain the system's initial *stakeholders* and interaction *invariants*:

stakeholder is a participant/actor of the system that arises from the early phases of requirements analysis. In general, a stakeholder comprises *knowledge*, being essentially a set of *knowledge items*. Formally, a stakeholder S is a tuple $\mathbb{S}\langle N, \mathcal{K}\rangle$ where N is the stakeholder's name and $\mathcal{K}$ its knowledge.

invariant is a system property that does not vary over time. Specifically, an invariant is a predicate over the knowledge of a set of stakeholders. The stakeholders are associated with the invariant by taking a *role* in it. A stakeholder takes a role in an invariant when a subset of its knowledge items is involved in the associated predicate.

More precisely, an invariant I is a tuple $\mathbb{I}\langle N, F, \mathcal{R}\rangle$, where: (i) N is the name of the invariant, (ii) F the predicate (formula), and (iii) $\mathcal{R}$ is a set of stakeholder roles referenced in F. Each role R, i.e., an element of $\mathcal{R}$, is a tuple of the form $\mathbb{R}\langle N_R, N_S, \mathcal{K}_{S\rightarrow I}, A\rangle$, where: (i) N_R is the role name, (ii) N_S the name of the associated stakeholder S, (iii) $\mathcal{K}_{S\rightarrow I}$ the knowledge of the stakeholder S that are involved in the associated invariant I, and (iv) A the cardinality of the role which can take either of the values $\{1, *\}$. Thus, a single stakeholder can take multiple roles (even providing/receiving the same set of knowledge items) in the same invariant. A role with cardinality 1 refers to exactly one stakeholder, while a role with cardinality $*$ refers to an unbounded set of stakeholders.

As an aside, the idea behind invariants is that in the system-to-be, there is a computation activity responsible for ensuring the validity of the invariant by changing the relevant knowledge of the stakeholders having a role in the invariant. Thus, it is possible to enrich a role definition by specifying which knowledge in the role does the stakeholder provide to/require from the invariant (its associated computation activity in particular). In this case, a role is a tuple of the form $\mathbb{R}\langle N_R, N_S, \mathcal{K}_{p:S\rightarrow I}, \mathcal{K}_{r:S\rightarrow I}, A\rangle$, where: (i) N_R is the role name, (ii) N_S the name of the associated stakeholder S, (iii) $\mathcal{K}_{p:S\rightarrow I}$ the knowledge the stakeholder S provides to the associated invariant I (its associated computation activity in particular), (iv) $\mathcal{K}_{r:S\rightarrow I}$ the knowledge the stakeholder S requires from the associated invariant I (its associated computation activity in particular), and (v) A the cardinality of the role which can take either of the values $\{1, *\}$.

Although one may think of invariants as (global) goals, the two terms are not always interchangeable. In fact, whereas goals are optative statements, typically referring to actions that will happen or be completed in the future, invariants have a more predicative nature, typically refering to the present state.

6.2.1 Invariant Decomposition

The system-level design process starts by identifying all top-level invariants, together with the stakeholders taking a role in them. Next, the process continues by decomposing the top-level invariants into sets of (sub-)invariants. Currently, such decomposition is always an AND-decomposition (rather than OR-decomposition). In order for the parent invariant to be satisfied, all of the decomposing sub-invariants must be satisfied simultaneously.

The decomposition of invariants is an iterative procedure that terminates once each leaf invariant in the decomposition tree is either of type *single-stakeholder* or *inter-stakeholder*. A single-stakeholder invariant is an invariant that references a single role only, i.e., it can be satisfied by manipulating the knowledge of a single stakeholder. On the other hand, an inter-stakeholder invariant is an invariant that references more than one role and can be satisfied only by exchange of knowledge items defined by the roles. Specifically, we consider only inter-stakeholder invariants that reference two roles, one with cardinality 1 and one with cardinality * (this is to reflect the coordinator-members relation of components in a DEECo ensemble, see Section 4.3).

Although inspired by the notion of goal decomposition in goal-oriented requirements acquisition [DvLF93], the objective here is not to provide a set of functional and non-functional requirements for the system-to-be. Rather, the decomposition steps should equally yield more insight regarding the requirements to be met and the relevant architecture schemes. At the same time, as the invariants get more specific, their definition should get more detailed (refined). At the end, the resulting invariants (leafs of the decomposition tree) should entail a precise description of stakeholder roles, including the knowledge provided/received by the associated stakeholders, as well as a specification of the invariant in terms of either manipulating knowledge of a single stakeholder or knowledge exchange.

With such decomposition, we strive to get to the level of abstraction, at which the invariants can be easily represented in the considered component communication semantics (for inter-stakeholder invariants) and component computation semantics (for single-stakeholder invariants).

6.2.2 Representation

The graphical representation of invariants, invariant decomposition, stakeholders and stakeholder roles is captured in a system-level graph (Figure 14). The direction of arrows in the *takes-role* relation indicates whether the stakeholder provides (pointing towards the invariant) and/or receives (pointing towards the stakeholder) knowledge to/from the invariant.

Further, a textual representation in the form of a DSL, providing additional information to the graphical representation, is available (as illustrated in Figure 15). Specifically, the DSL adds a detailed description of the roles (i.e., defines a full list of provided/received knowledge; e.g., lines 5, 23), the invariant's textual description (e.g., line 7), its formal specification (e.g., lines 28-29), and its decomposition (e.g., line 9). The formal specification defines the semantics of the invariant in a predicate logic notation.

There are also similar graphical and textual representations for the concepts on the other design levels (i.e., ensemble and component level). We omit their description, since it is out of scope of this document.

6.2.3 Cloud Load Balancing Example

In order to illustrate the above-defined concepts, we elaborate on a simple example scenario from the Science Cloud Case Study [SRA+11]. In this scenario, several network nodes, forming an open-ended cloud platform, run third-party services. Provided there exists an external mechanism to migrate a

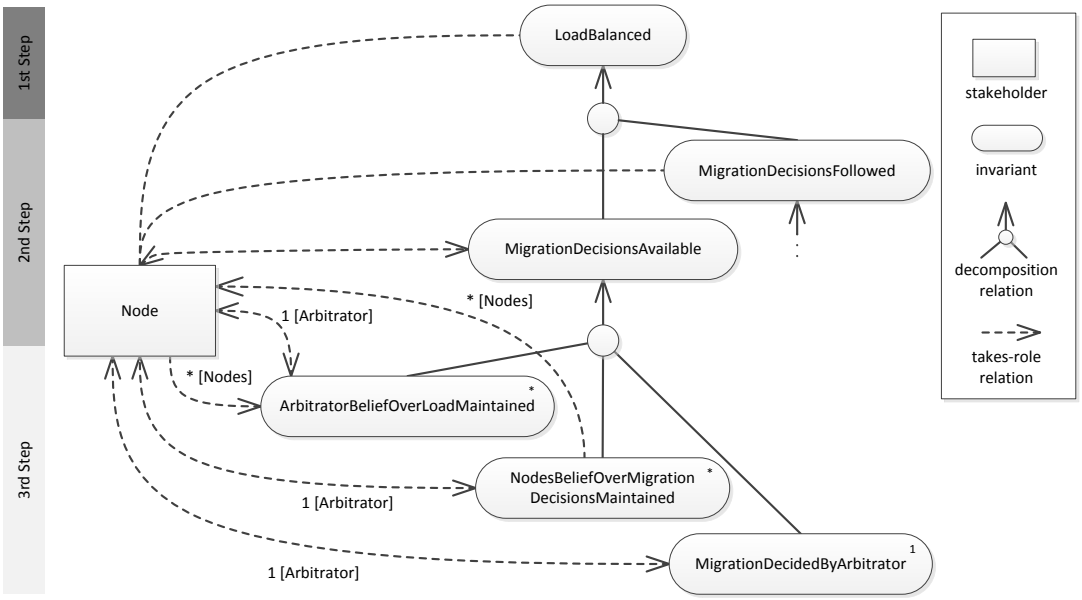

Figure 14: System-level design example.

service from one node to another, the goal is to keep the load evenly balanced among the participating nodes.

Figure 14 shows a system-level diagram for the scenario. In this case, the invariant decomposition was performed in three steps. In particular, defined in the first step, the top-level invariant LoadBalanced is, in the second step, decomposed into two invariants MigrationDecisionsAvailable and MigrationDecisionsFollowed, which themselves are subject to decomposition (the third step). As one of the possible ways, the MigrationDecisionsAvailable is decomposed into ArbitratorBeliefOverLoadMaintained, NodesBeliefOverMigrationDecisionsMaintained and MigrationDecidedByArbitrator.

The decomposition in the second step reflects the design choice to separate the part responsible for deciding migration from the migration process itself, in order to achieve separation of concerns. Similarly, the decomposition of MigrationDecisionsAvailable in the third step reflects an approach to deciding on migration where a (arbitrary, dynamically chosen) Node with the role of Arbitrator must gather the information about every node's load, take migration decisions, and broadcast them to other Nodes. The main idea of this approach is to allow for centralized decision-making, while contributing to robustness by selecting the decision maker dynamically.

Upon closer inspection, we can conclude that no further decomposition of these three invariants is necessary, as the former two are *inter-stakeholder*, whereas the latter one is a *single-stakeholder* invariant.

Figure 15 shows the DSL representation of some of these invariants, in particular those which are in Figure 14 decomposed to the level of leaves. As the invariants get more specific and detailed in each consecutive decomposition step, so does their textual and graphical representation. LoadBalanced definition (lines 3-9), for example, does not contain a formal specification, neither states the knowledge the stakeholders (here Nodes) have to provide or receive. In comparison, the definition of e.g. Arbitra-

```
1  stakeholder Node
2
3  invariant LoadBalanced:
4      stakeholders:
5          ∗ Node as Nodes  % <cardinality> <stakeholder> as <role name>
6      description:
7          "The overall load of Nodes in the same local network is balanced"
8      decomposition:
9          MigrationDecisionsAvailable and MigrationDecisionsFollowed
10
11 invariant MigrationDecisionsAvailable:
12     stakeholders:
13         ∗ Node as Nodes providing {NetworkID, LoadRatio} receiving {MigrationDecisions}
14     description:
15         "Nodes have correct and up−to−date migration decisions"
16     decomposition:
17         ArbitratorBeliefOverLoadMaintained
18         and MigrationDecidedByArbitrator
19         and NodesBeliefOverMigrationDecisionsMaintained
20 ...
21 invariant ArbitratorBeliefOverLoadMaintained:
22     stakeholders:
23         1 Node as Arbitrator providing {NetworkID} receiving {NodesToMigrateIn, NodesToMigrateOut},
24         ∗ Node as Nodes providing {NetworkID, LoadRatio}
25     description:
26         "Arbitrator has a correct belief over nodes requiring/allowing migration in the same network."
27     formal_specification:
28         Arbitrator.NodesToMigrateIn = {n | n ∈ Nodes ∧ n.NetworkID = Arbitrator.NetworkID ∧ n.LoadRatio ≤
               MIN_LOAD}
29         Arbitrator.NodesToMigrateOut = {n | n ∈ Nodes ∧ n.NetworkID = Arbitrator.NetworkID ∧ n.LoadRatio ≥
               MAX_LOAD}
30
31 invariant MigrationDecidedByArbitrator:
32     stakeholders:
33         1 Node as Arbitrator providing {NodesToMigrateIn, NodesToMigrateOut} receiving {MigrationDecisions}
34     description:
35         "Arbitrator decides the migration based on its belief over the other nodes' load."
36     formal_specification:
37         Arbitrator.MigrationDecisions = decideMigration(Arbitrator.NodesToMigrateIn, Arbitrator.
               NodesToMigrateOut)
38 ...
```

Figure 15: System-level entities definition example in DSL

torBeliefOverLoadMaintained (lines 21-29) contains both a formal specification and provided/received knowledge description.

6.3 Ensemble Level

At the ensemble level, the goal is to refine

- the inter-stakeholder invariants identified at the end of system-level analysis by means of *ensembles*, and
- the roles of stakeholders by means of *interfaces*.

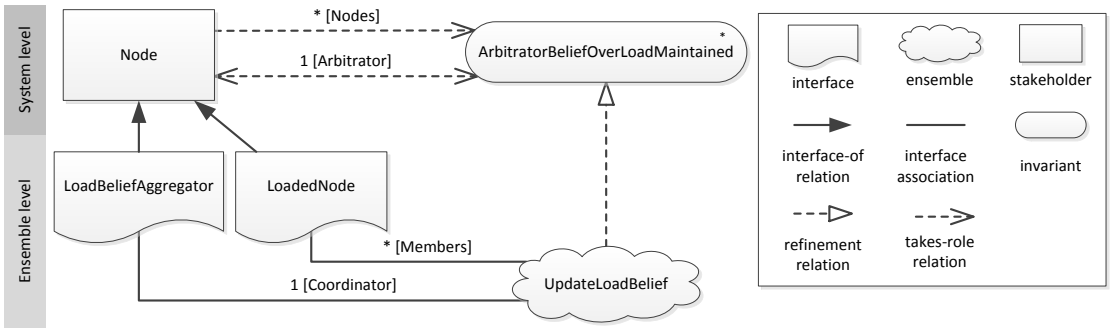

Figure 16: Ensemble-level diagram example.

6.3.1 Interface Identification

An *interface* entails the knowledge items that a stakeholder needs to provide to/receive from the associated inter-stakeholder invariant. Formally, an interface $\mathcal{I}$ is a tuple $\mathbb{AI}\langle N_{\mathcal{I}}, \mathcal{K}\rangle$, where $N_{\mathcal{I}}$ is the name of the interface and $\mathcal{K}$ the set of the entailed knowledge items. Thus an interface refines a stakeholder's role in an invariant by stating the relevant knowledge items.

In Figure 17 LoadedBeliefAggregator and LoadedNode interfaces refine the role of Node in ArbitratorBeliefOverLoadMaintained (Figure 15, lines 23-24).

6.3.2 Inter-stakeholder Invariant Refinement

Each of the inter-stakeholder invariants identified at the system level is refined by an ensemble (multiple invariants can be refined by the same ensemble). As described in Section 4.3, an *ensemble* is a group of components, where both the *membership* in the group and the communication in the group, in the form of *knowledge exchange*, are expressed declaratively. Specifically, one component of the group is the *coordinator* of the group, while the other components are *members*; knowledge exchange is allowed only between the coordinator and a member of the ensemble. The coordinator and members are identified via their interfaces. Formally, an ensemble E is a tuple $\mathbb{E}\langle N_E, \mathcal{I}_C, \mathcal{I}_M, M, X\rangle$, where: (i) N_E is the name of the ensemble, (ii) $\mathcal{I}_C$ is the coordinator interface (iii) $\mathcal{I}_M$ is the member interface, (iv) M is the membership predicate, i.e., a function $\mathcal{K}_{\mathcal{I}_C} \times \mathcal{K}_{\mathcal{I}_M} \rightarrow \{true, false\}$, and (v) X the knowledge exchange definition, i.e., a function $\mathcal{K}_{\mathcal{I}_M} \times \mathcal{K}^*_{\mathcal{I}_M} \rightarrow \mathcal{K}_{\mathcal{I}_C} \times \mathcal{K}^*_{\mathcal{I}_C}$, where $\mathcal{K}_{\mathcal{I}}$ is the set of relevant knowledge items of the interface $\mathcal{I}$ and * refers to an unbounded number of members of the ensemble.

The membership and knowledge exchange of an ensemble (i.e., M and X) are to be inferred from the invariant's formal or textual specification. The member and coordinator interfaces are determined from the cardinality of roles. Specifically, the role with cardinality 1 is refined into the coordinator interface, while the one with cardinality * is refined into the member interface.

Figure 16 is an example of ensemble-level diagram. ArbitratorBeliefOverLoadMaintained invariant (Figure 15) is refined into the UpdateLoadBelief ensemble. The particular ensemble definition in Figure 17, which supplements the diagram, states that the ensemble is formed only when member and coordinator have the same NetworkID (membership). In that case, NodesToMigrateIn and NodesToMigrateOut lists are populated (knowledge exchange) with member ids that are under- and over-loaded respectively.

```
1  interface LoadBeliefAggregator of Node:
2      NetworkID, NodesToMigrateIn, NodesToMigrateOut     % elaborates Figure 15, line 23
3
4  interface LoadedNode of Node:
5      NetworkID, LoadRatio                               % elaborates Figure 15, line 24
6
7  ensemble UpdateLoadBelief refines ArbitratorBeliefOverLoadMaintained:
8      coordinator: LoadBeliefAgregator                   % determined by cardinality 1 in Figure 15, line 23
9      member: LoadedNode                                 % determined by cardinality * in Figure 15, line 24
10     membership:                                        % elaborates Figure 15, lines 28–29
11         coordinator.NetworkID = member.NetworkID
12     knowledge_exchange:                                % elaborates Figure 15, lines 28–29
13         coordinator.NodesToMigrateIn ← members.filter(member.LoadRatio ≤ MIN_LOAD)
14         coordinator.NodesToMigrateOut ← members.filter(member.LoadRatio ≥ MAX_LOAD)
15     scheduling: periodic( 100ms )                      % elaborates Figure 15, lines 28–29
```

Figure 17: Ensemble-level DSL example: refinement of ArbitratorBeliefOverLoadMaintained

6.4 Component Level

At the component level, the goal is threefold: (i) to refine a stakeholder by means of a *component* (component knowledge in particular), (ii) to refine the single-stakeholder invariants that the stakeholder takes a role in by means of the component's *processes*, and (iii) to reify the interfaces of the stakeholder, defined during the ensemble-level design phase, by means of the component's knowledge. Naturally, there may be several ways to refine a stakeholder, thus selecting a particular variant is a necessary design decision. According to Section 4.2, a component consists of *knowledge* and *processes* (operating solely upon the knowledge). Formally, a component C is a tuple $\mathbb{C}\langle N_C, \mathcal{K}, \mathcal{P}\rangle$, where: (i) N_C is the name of the component, (ii) $\mathcal{K}$ the knowledge – a set of knowledge items (recall Section 6.2), and (iii) $\mathcal{P}$ the set of processes. A process P is formally a tuple $\mathbb{P}\langle N_P, \mathcal{K}_i, \mathcal{K}_o, F\rangle$, where: (i) N_P is the name of the process, (ii) $\mathcal{K}_i$ the input knowledge of the process, (iii) $\mathcal{K}_o$ the output knowledge of the process, and (iv) F the function of the process (i.e., a function $\mathcal{K}_i \rightarrow \mathcal{K}_o$).

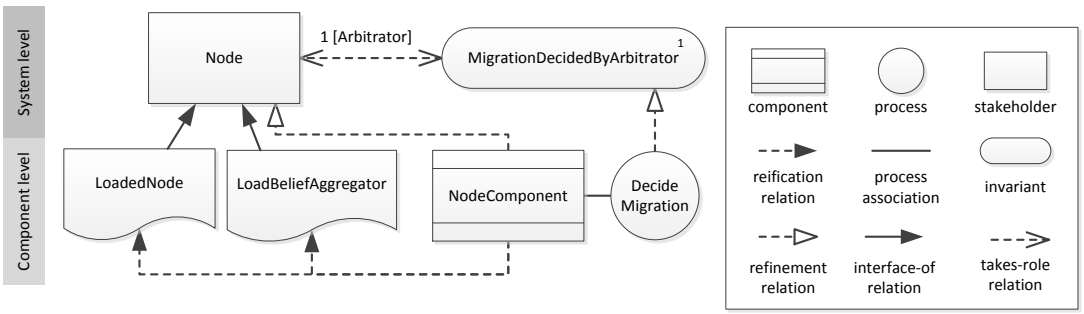

Figure 18: Component-level design example.

6.4.1 Stakeholder Refinement

In general, a stakeholder is refined by one or more *components*, while a single component can refine several stakeholders. The main goal of such refinement is to aggregate all the knowledge items the stakeholder provides to/receives from all the single-stakeholder invariants the stakeholder takes a role in.

In Figure 18, the component NodeComponent refines the Node stakeholder taking a role into the MigrationDecidedByArbitrator invariant (Figure 15). Thus, the component must include all the related knowledge items (i.e., NodesToMigrateIn, NodesToMigrateOut, and MigrationDecisions) as shown in Figure 19.

6.4.2 Single-stakeholder Invariant Refinement

Every single-stakeholder invariant is refined by a local computation activity – a process – of the component refining the associated stakeholder. The input/output knowledge of the process is determined by the knowledge items provided to/received from the invariant by the associated stakeholder. The other aspects of the process, i.e., the function performing the computation and scheduling, are designed manually, following the description/formal specification of the invariant.

In Figure 18, the process DecideMigration of NodeComponent refines the MigrationDecidedByArbitrator invariant (Figure 15). Thus, its input constitutes the NodesToMigrateIn and NodesToMigrateOut knowledge items, while its output comprises the MigrationDecisions knowledge item, as shown in Figure 19. The ComputeDecisions function employed by the process represents the procedure to compute the migration decisions according to the formal specification given in Figure 15.

6.4.3 Interface Reification

A component refining a stakeholder must also reify all its associated interfaces (in that case we say that the component *features* the interfaces). A reification of an interface implies including all the knowledge items specified in the interface.

In Figure 18, the component NodeComponent reifies the LoadBeliefAggregator and LoadedNode (and potentially other) interfaces (Figure 17), since the associated Node stakeholder takes a role in the ArbitratorBeliefOverLoadMaintained invariant (Figure 15). Thus, the component must include the NetworkID, LoadRatio, NodesToMigrateIn, and NodesToMigrateOut knowledge items, as shown in Figure 19.

6.5 Formalization of Requirements

The design method utilizes predicates to capture requirements. It does not presume any particular form of predicates – the predicates can be articulated informally as plain English sentences or in a formal language (this can be combined at several levels of detail). In this context, SOTA and Poem (D4.2, [ZAC+12]) are planned as the formal languages for concretizing the requirements, while GEM provides an underlying logical framework for formalization of requirements and adaptation. Their use would bring an additional benefit of automated reasoning on the requirements and the option of analyzing component/ensemble fitness in different environments. Since SOTA features adaptation patterns, this would potentially help guide the application designer in addressing typical adaptation scenarios.

```
1  component NodeComponent refines Node features LoadBeliefAggregator, LoadedNode, ... :
2      knowledge:                                          % elaborates Figure 15, lines 23, 24, 33
3          NetworkID: IPAddress = "10.10.1.x",
4          LoadRatio: float = ... ,
5          NodesToMigrateIn: list NodeID = ... ,
6          NodesToMigrateOut: list NodeID = ... ,
7          MigrationDecisions: map NodeID → NodeID = ... ,
8          ComputeDecisions: fun ... ,
9          ...
10     process DecideMigration refines MigrationDecidedByArbitrator:
11         input: { NodesToMigrateIn, NodesToMigrateOut }   % elaborates Figure 15, line 33
12         output: { MigrationDecisions }                   % elaborates Figure 15, lines 33
13         function: ComputeDecisions                       % elaborates Figure 15, lines 37
14         scheduling: periodic( 100ms )                    % elaborates Figure 15, lines 37
15     ...
```

Figure 19: Component-level entities example: refinement of MigrationDecidedByArbitrator

6.6 Knowledge Representation

The high-level design method extensively builds on knowledge specification, both in stakeholders and invariants, as well as in the later phases during detailed design of ensembles and components. In particular, it assumes that the detailed description of knowledge and its semantics will be formalized using KnowLang (D3.2, [VHM+12]) and that it will provide the formal framework and tools for supporting systematic elaboration of stakeholder knowledge representation. Further, building on the features of the KnowLang Reasoner, we plan to enrich the stakeholder knowledge and invariant specification by means of derived and uncertain knowledge. As for modeling self-adaptive behavior, we envision to integrate the KnowLang behavioral concepts (i.e., situation and policy) by building on top of the more general concept of component process. Such integration would help filling the abstraction gap between the conceptual design on the system level and the detailed design on the component level.

6.7 Experiments with the Design Method in the E-mobility Case Study

In cooperation with VW, the high-level design method is currently subject to an extensive experimental work focused on the e-mobility case study, where a method for identifying ensembles is inherently needed. Initial results are reported in D7.2 [S+12]. Based on these preliminary results, we plan to create a second iteration, where we intend to elaborate in detail on all the identified ensembles and components along with their knowledge. In this respect, the use of SOTA for formalizing the requirements during the gradual requirements refinement is a promising option. In the same vein, we plan experimenting with KnowLang for identifying factual knowledge and managing the uncertain and derived knowledge. Further, we envision to investigate and quantify the impact of communication and computation delays on the overall conformance of the application to the initial requirements.

7 On-going Work on Interpreting Performance as Knowledge

As a follow-up to the implementation-level language extensions (jRESP and DEECo), in this section we present initial results in interpreting the observed performance as a part of the component knowledge. The benefit of such interpretation is two-fold. Firstly, having the observed performance available in the form of component knowledge allows for conformance checking of performance-related prop-

erties of component and ensemble implementation. Furthermore, the autonomic behavior of components and ensembles may depend on the observed performance of both the components and ensembles themselves, and other parts of the system. It may also be useful for the ensemble membership to be based on performance indicators.

From the "surrounding environment" point of view, performance knowledge is just another tuple where the value represents processing time, load factor or another performance indicator. From the component point of view, however, performance knowledge is different from other knowledge tuples. First, accessing the performance knowledge may involve an inherent cost that is not necessarily present with other knowledge types: the actual performance must be measured and the measured data must be processed. This cost is typically paid by the observed rather than the querying component, because it is that component that must be measured or measure itself. In multiple senses (robustness, precision, timeliness), the cost of measurement also stands as a tradeoff against accuracy.

Also important is the observation that performance knowledge obtained through measurement is necessarily inaccurate. In part, this property resembles that of other knowledge, such as the information from hardware sensors. Additionally, however, the measurement code can directly affect the application, resulting for example in the system reaching different optimization decisions.

To mitigate some of the problems related to interpreting performance as knowledge, we introduce the Stochastic Performance Logic, which allows for working with component and ensemble performance data while abstracting away from some of the performance data properties listed above.

7.1 Stochastic Performance Logic

Stochastic Performance Logic (SPL) is a many-sorted first-order logic with relational operators between performance indicators [BBK+12]. These operators allow one to express relative comparisons of performance, such as testing whether component A is faster than component B.

SPL operates on random variables representing performance of a component. The relational operators accept such random variables as their input, and produce an output indicating whether the relation is true based on the performance data provided. In mathematical terms, this amounts to testing hypotheses over the properties of the random variables, and SPL relies on statistical procedures to evaluate the operators.

With further extensions to SPL and with integration of SPL with multiple data sources, as introduced in [BBH+12], the complex issues of accuracy and its relationship to the measurement cost become more manageable. First, transparent to the developer, reasoning about performance takes the robust form of statistical testing, and the probabilities of certain types of errors due to random noise can be managed. Next, we separate the evaluation of SPL formulas from the data collection process, introducing the possibility of extrapolating from past measurements with negligible increase of SPL formula complexity. The extrapolation not only makes it possible to manage the overhead of an on-demand measurement, but can also predict future trends, thus helping the ensembles to adapt ahead of time.

7.2 Runtime Framework for Collecting Performance Knowledge

At the tool level, the the mechanisms to apply SPL in the ensemble prototypes are implemented by the SPL framework, currently under development. The SPL framework consists of two main parts, one that derives knowledge from the measured performance data, and one that takes care of actually measuring the performance. Both parts need to be integrated with the ensemble runtime environments. Thus, the current prototype focuses on Java, but otherwise remains at a platform neutral level.

During the course of the project, the framework has already been tested with an OSGi component framework as a preparation step for integration into the Scientific Cloud Platform, itself under develop-

ment. The test within an OSGi framework is important because the OSGi class-loading mechanism, which needs to interact with the instrumentation code for measurement, is different from standard class loading procedures. The difference is that standard class loading is based on a tree hierarchy of class loaders, while OSGi components use a set of independent class loaders to allow a complete isolation of individual components. The test shows that it is possible to add SPL framework even to such non-standard environment, and that the integration is possible without introducing extra burden to the user of the framework.

7.3 Techniques for Obtaining Performance Information

There are multiple options how to obtain the performance information in the context of components and ensembles. Traditionally, the developer of an ensemble component can manually write the code that publishes the information. This approach is useful in that it allows one to handle special cases in a flexible way, where the mapping between the ensemble behavior and the observable events is complex. On the other hand, this solution burdens the developer and is static in nature. It is therefore necessary to complement it with an automated and dynamic performance measurement support.

To provide the performance measurement support, we rely on the connection to the implementation level mapping. We observe that many of the events whose performance properties are of interest are directly mapped to well defined code locations, such as method invocation. When it makes sense to insert the measurement code at such code locations, it is possible to automate this process.

In the DEECo framework all events are mapped to method invocation. For example, implementation of the *process* is actually a method annotated with `@DEECoProcess`. Similarly, other events (e.g. knowledge transfer) are mapped to Java methods. This allows us to provide the automation mentioned above.

To fully separate implementation of the components or ensembles from the measuring framework, the instrumentation – insertion of the measurement code – shall work at bytecode level to eliminate needs for changing existing source code. Java offers support for bytecode instrumentation, typically by deploying a Java agent that transforms (e.g. instruments) the bytecode. But the instrumentation in Java can happen only at class loading time – typically when an application is starting. Recent Java Virtual Machines (JVM) make it possible to force class *reloading*: therefore the instrumentation can happen at virtually any time during execution. This allows one to add or remove the measurement code at will during ensemble or component execution, providing dynamic performance measurement support.

Java itself only provides a very basic API to modify bytecode. For efficient progress, we have employed the DiSL instrumentation framework [MZA+12] that offers an elegant way to instrument bytecode. The DiSL framework can insert the instrumentation code to various code locations, ranging from basic blocks (such as loop starts) to method bodies. The framework was designed for static instrumentation, whereby the instrumentation happens during application start and the application remains instrumented until terminated.

In our scenario, we are using DiSL in a very different setting. The novelty is in using it in a very dynamic environment: first, the component or ensemble instrumentation can happen at any time during its execution. Next, it shall be possible to remove the instrumentation. The DiSL framework itself does not offer the mentioned functionality, but extending it is under development. With these new features, DiSL would be used as the instrumentation framework that allows for both dynamic and automatic support to obtain the performance measurement.

8 Conclusion and Outlook

In this document, we described two SCEL implementations – jRESP and DEECo (realized in Java by jDEECo) – as the foundation for implementation-level conformance checking. They both allow for the use of SCEL paradigms in Java, but differ in their focus. While jRESP aims at rapid prototyping and experiments with SCEL, DEECo/jDEECo targets large-scale system development with well-defined architecture. This is further supported by the presented design method that allows distilling a DEECo-flavoured architecture of components and ensembles from system requirements.

With respect to the on-going work, in accordance with the project plan, the focus will be on verification of functional properties of systems implemented in jDEECo. This is to be realized via an extension of JPF as already outlined in this report. Additionally, we plan to refine the memory and threading model of the the jDEECo framework, and further extend JPF to fully exploit the specific features of jDEECo-based systems, which have the promising potential to make verification more efficient. With respect to performance as knowledge, we plan to provide techniques for SPL-conformance checking at runtime.

References

[BBH+12] Lubomir Bulej, Tomas Bures, Vojtech Horky, Jaroslav Keznikl, and Petr Tuma. Performance Awareness in Component Systems: Vision Paper. In *Proceedings of COMPSAC 2012*, COMPSAC '12, 2012.

[BBK+12] Lubomir Bulej, Tomas Bures, Jaroslav Keznikl, Alena Koubkova, Andrej Podzimek, and Petr Tuma. Capturing Performance Assumptions using Stochastic Performance Logic. In *Proc. 3rd Intl. Conf. on Performance Engineering*, ICPE'12, Boston, MA, USA, 2012.

[BGG+03] Paolo Bresciani, Paolo Giorgini, Fausto Giunchiglia, John Mylopoulos, and Anna Perini. Tropos: An agent-oriented software development methodology, 2003.

[CHK+12] Jacques Combaz, Vojtech Horky, Jan Kofron, Jaroslav Keznikl, Alberto Lluch Lafuente, Michele Loreti, Philip Mayer, Carlo Pinciroli, Petr Tuma, and Andrea Vandin. The SCE Workbench and Integrated Tools, Pre-Release 1. ASCENS Deliverable D6.2, October 2012.

[DFLP11] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese. Languages primitives for coordination, resource negotiation, and task description. ASCENS Deliverable D1.1, September 2011. `http://rap.dsi.unifi.it/scel/`.

[DFLP12] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese. A language-based approach to autonomic computing. In *Proc. of the 10th International Symposium on Software Technologies Concertation on Formal Methods for Components and Objects (FMCO 2011)*, Lecture Notes in Computer Science. Springer, 2012. To appear.

[DvLF93] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Sci. Comput. Program.*, 20(1-2):3–50, April 1993.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7:80–112, 1985.

[HBGK12] Matthias Hölzl, Lenz Belzner, Thomas Gabor, and Annabelle Klarl. The ASCENS Service Component Repository (first version). ASCENS Deliverable D8.2, October 2012.

[JPF] Java PathFinder.
http://babelfish.arc.nasa.gov/trac/jpf/.

[KBPK12] Jaroslav Keznikl, Tomas Bures, Frantisek Plasil, and Michal Kit. Towards Dependable Emergent Ensembles of Components: The DEECo Component Model. In *Proceedings of WICSA/ECSA 2012*. IEEE, August 2012.

[MZA+12] Lukáš Marek, Yudi Zheng, Danilo Ansaloni, Walter Binder, Zhengwei Qi, and Petr Tuma. DiSL: An extensible language for efficient and comprehensive dynamic program analysis. In *Proc. 7th Workshop on Domain-Specific Aspect Languages*, DSAL '12, pages 27–28, New York, NY, USA, 2012. ACM.

[PBN+12] Rosario Pugliese, Tomas Bures, Rocco De Nicola, Jaroslav Keznikl, Michele Loreti, Frantisek Plasil, and Francesco Tiezzi. Languages for Coordinating Ensemble Components. ASCENS Deliverable D1.2, October 2012.

[PTO+11] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, Timothy S. Stirling, Álvaro Gutiérrez, Luca Maria Gambardella, and Marco Dorigo. Argos: A modular, multi-engine simulator for heterogeneous swarm robotics. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2011, San Francisco, CA, USA, September 25-30, 2011*, pages 5027–5034. IEEE, 2011.

[S+12] Nikola Serbedzija et al. Ensemble Model Syntheses with Robot, Cloud Computing and e-Mobility. ASCENS Deliverable D7.2, October 2012.

[SRA+11] Nikola Serbedzija, Stephen Reiter, Maximilian Ahrens, Jose Velasco, Carlo Pinciroli, Nicklas Hoch, and Bernd Werther. Requirement specification and scenario description of the ascens case studies. ASCENS Deliverable D7.1, 2011.

[VHM+12] Emil Vassev, Mike Hinchey, Ugo Montanari, Nicola Bicocchi, and Franco Zambonelli. The KnowLang Framework for Knowledge Modeling for SCE Systems. ASCENS Deliverable D3.2, October 2012.

[vL01] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour, 2001.

[ZAC+12] Franco Zambonelli, Dhaminda B. Abeywickrama, Giacomo Cabri, Mariachiara Puviani, Matthias Hoelzl, Andrea Corradini, Alberto Lluch Lafuente, and Rocco De Nicola. Component- and Ensemble-level Self-Expression Patterns: Report on Experimental and Simulation Activities, and Requirements for Tools Implementation. ASCENS Deliverable D4.2, October 2012.