Design of Ensemble-Based Component Systems by Invariant Refinement

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

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

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

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

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

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

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

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

[3]Corporate Research Group
Volkswagen AG
Wolfsburg, Germany

ABSTRACT

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

Categories and Subject Descriptors

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

Keywords

Component; ensemble; refinement; requirements engineering; system design

1. INTRODUCTION

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

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

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

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

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

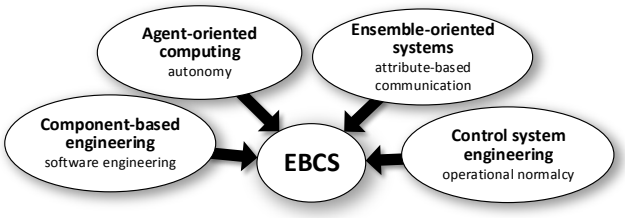

Figure 1: Context of Ensemble-Based Component Systems (EBCS).

2. ENSEMBLE-BASED COMPONENT SYSTEMS: A CASE STUDY

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

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

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

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

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

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

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

```
interface AvailabilityAggregator:
    calendar, availabilityList

interface AvailabilityAwareParkingLot:
    position, availability

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

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

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

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

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

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

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

2.2 Illustration of the Concepts on the Case Study

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

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

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

3. PROBLEM STATEMENT

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

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

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

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

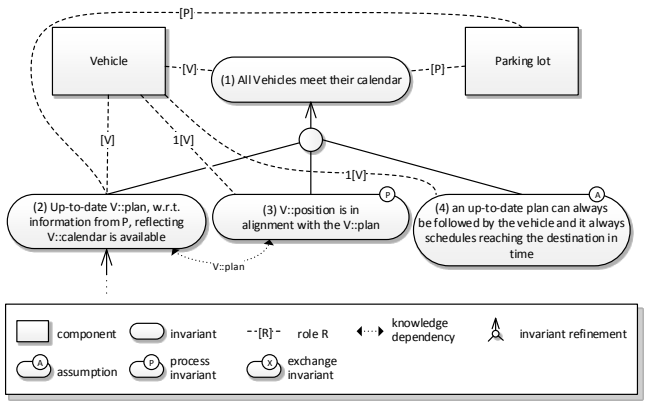

Figure 3: Top-level design of the case study.

4. DESIGNING ENSEMBLES VIA INVARIANT REFINEMENT

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

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

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

4.1 Invariants and Assumptions

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

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

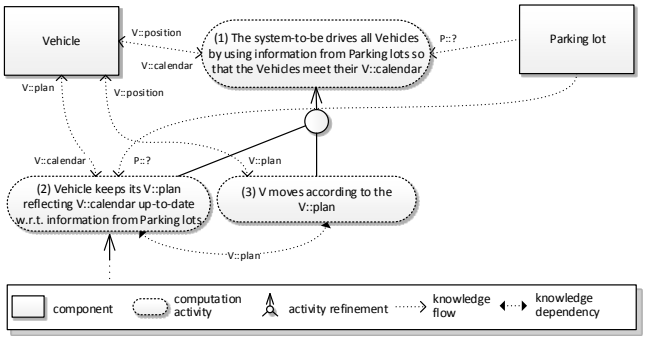

Figure 4: Dual, computation-activity-based view on the top-level design of the case study from Figure 3.

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

4.2 Invariants vs. Computation Activities

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

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

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

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

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

4.3 Invariant Refinement

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

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

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

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

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

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

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

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

4.4 Leaves of Refinement

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

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

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

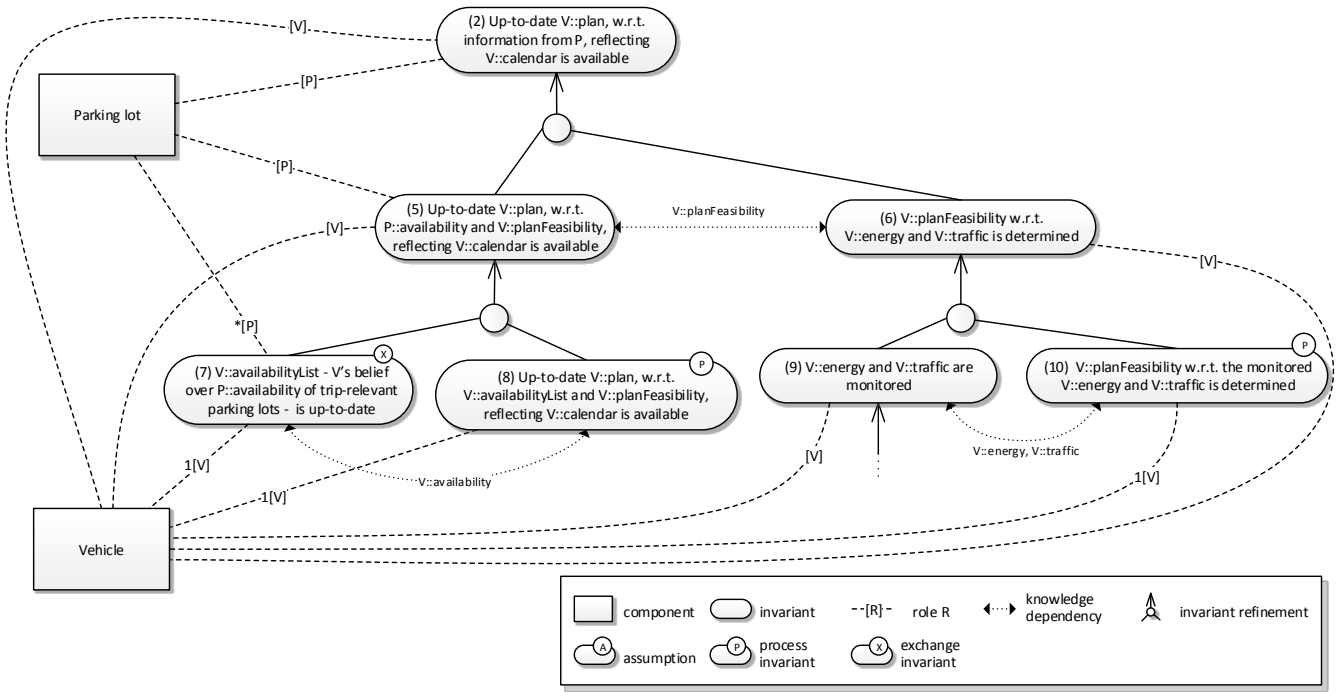

Figure 5: Invariant refinement of "V has an up-to-date V::plan reflecting V::calendar".

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

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

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

4.5 From Invariants to Final Architecture

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

5. BRIDGING ABSTRACTION LEVELS VIA INVARIANT PATTERNS

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

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

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

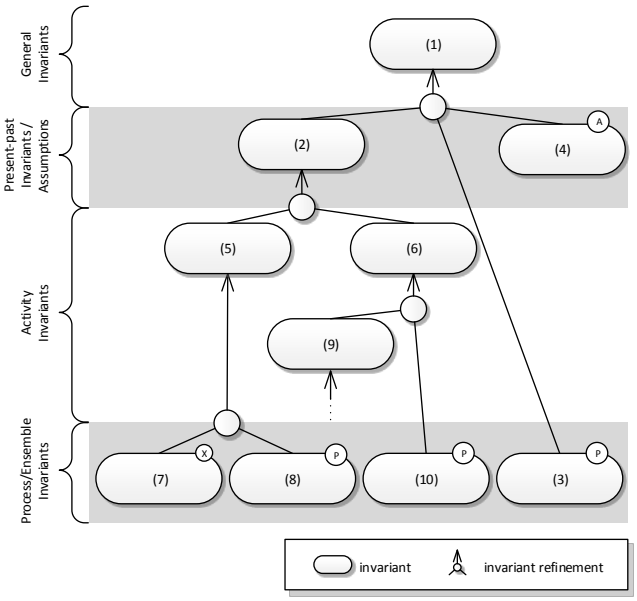

Figure 6: Patterns of invariants in the case study.

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

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

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

5.1 General Invariants

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

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

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

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

5.2 Present-past Invariants

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

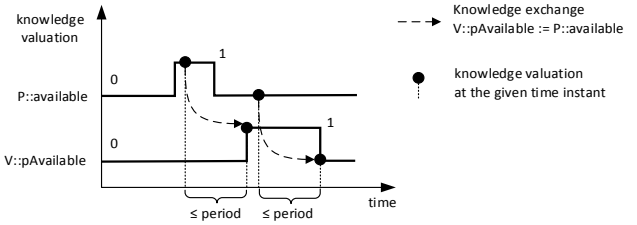

Figure 7: Example of knowledge evolution in time when employing (periodic) knowledge exchange.

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

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

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

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

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

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

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

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

5.3 Activity Invariants

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

Being relatively low-level, an activity invariant reflects detailed properties of a computation activity that corresponds to software computation. First, it captures the requirement that the output knowledge changes only as a result of the computation activity. Here, we assume that no activities have the same output knowledge. Moreover, an activity invariant captures read consistency of the input knowledge, i.e., that each output

knowledge valuation is based on the same or newer input knowledge valuation than the previous one. In an ideal case, the computation is instant, relating thus the current valuation of both the input and output knowledge. Similarly to present-past invariants, the maximal distance in the past needed to formulate the operational normalcy is expressed by the lag of the invariant.

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

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

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

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

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

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

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

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

5.4 Process invariants

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

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

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

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

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

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

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

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

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

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

5.5 Ensemble invariants

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

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

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

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

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

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

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

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

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

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

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

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

5.6 Refinement among Invariant Patterns

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

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

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

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

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

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

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

6. EVALUATION AND DISCUSSION

6.1 Case Study

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

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

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

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

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

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

6.2 Correctness by Construction

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

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

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

6.3 Runtime Verification

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

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

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

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

6.4 Novelty and Benefits

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

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

7. RELATED WORK

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

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

8. CONCLUSION AND FUTURE WORK

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

IRM raises a number of interesting questions for further research. In particular, they include: (i) providing a formal framework (i.e.,

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

9. ACKNOWLEDGMENTS

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

10. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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