The Invariant Refinement Method*

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

Charles University in Prague,
Faculty of Mathematics and Physics,
Department of Distributed and Dependable Systems,
Prague, Czech republic

Abstract. The chapter describes IRM, a method that guides the design of smart-cyber physical systems that are built according to the autonomic service-component paradigm. IRM is a requirements-oriented design method that focuses on distributed collaboration. It relies on the invariant concept to model both high-level system goals and low-level software obligations. In IRM, high-level invariants are iteratively decomposed into more specific sub-invariants up to the level that they can be operationalized by autonomous components and component collaborations (ensembles). We present the main concepts behind the method, as well the main decomposition patterns that back up the design process, and illustrate them in the ASCENS e-mobility case study.

Keywords: system design, dependability, self-adaptivity

1 Introduction

Business needs and technological breakthroughs have been recently pushing towards the cost-effective and manageable development of increasingly complex, software-intensive systems that feature close connection to the physical world – so-called smart *cyber-physical systems* (CPS). Examples of such systems are numerous: smart electric grids, emergency coordination systems, autonomous robots, fleets of cooperating vehicles, smart spaces, to name just a few.

Within the ASCENS project, we have created a comprehensive software engineering solution for the development of smart CPS. The solution takes the form of a framework consisting of:

(i) a specialized software component model, based on the paradigm of *autonomic component ensembles* (ACEs), with clear execution and interaction semantics;
(ii) an execution environment that allows for distributed and decentralized operation of systems composed of the specialized software components;
(iii) design-time and runtime analysis (e.g., timing analysis) based on a well-defined computational model; and

* This research was supported by the European project IP 257414 (ASCENS).

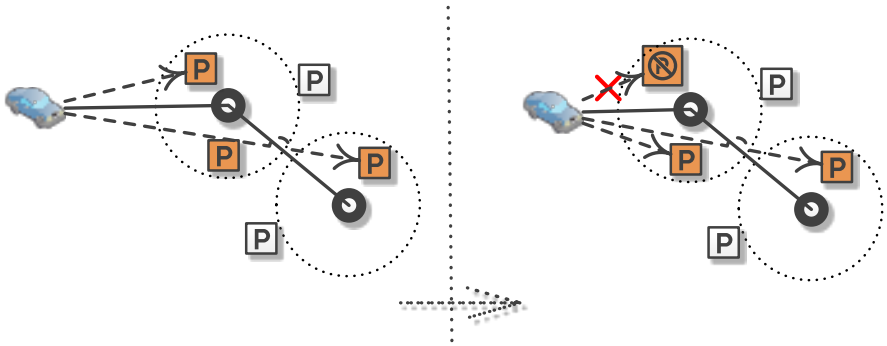

Fig. 1. E-mobility case study: electric cars need to proactively re-plan according to the availability of parking stations.

(iv) a specialized requirements-oriented design method that focuses on distributed collaboration and complements (i).

In this chapter, we focus on the last element of our framework and present the design method and associated model – the *Invariant Refinement Method* (IRM). IRM features contractual design based on the iterative refinement of system-level requirements, and provides both *dependability* in form of traceability of software artifacts to system-level goals, and *adaptability*, as it captures the design alternatives pertaining to different operational contexts/situations and translates them into different system and component modes. From the overall perspective of the Ensemble Development Life Cycle (see Chapter III.1 [13]), IRM thus serves as a method to guide the transition from early high-level requirements (featured by SOTA/GEM) to software architecture of autonomic components and ensembles.

The chapter is based on the authors' papers [7,10,15] and technical reports [6] and is structured as follows. Section 2 presents our running example and illustrates the basic principles behind our ACEs-based component model. Section 3 details on the limitations of traditional software engineering methods when designing CPS via ACEs. Section 4 presents the basics of IRM, while Sections 5 details on the specific refinement patterns that can be employed in the IRM-based design. Finally, Section 6 concludes the chapter and discusses the yet-to-be-addressed challenges.

2 Running Example

To illustrate the IRM approach, we use a scenario taken from the ASCENS e-mobility case study (Chapter IV.4 [12]). In this case study, a fleet of *electric vehicles* (e-vehicles) is used to distribute people to their *places of interest* (POIs) in a city. Due to their limited autonomy compared to conventional vehicles, e-vehicles need to regularly stop at parking lots with energy charging capabilities

located in designated parking stations in the city. After recharging, e-vehicles become again fully operational and join the rest of the fleet.

Careful planning is needed in order to avoid traffic bottlenecks and high recharging times. The problem in such planning is threefold:

(i) The whole system is very dynamic, as vehicles change their routes according to the passengers calendars (which can also change at runtime), streets/parking stations can be temporarily closed, and the load of parking stations is typically hard to predict as it changes according to the incoming parking requests from the vehicles (which change as vehicles re-plan).
(ii) No central communication and coordination point is assumed. This results into having an inherently scalable system which is harder to control, as each vehicle plans its own route according to its partial view of the rest of the system and independently of the rest of the vehicles.
(iii) Each element of the system may be in different modes (e.g., "low battery" vs. "fully operational" for the e-vehicles) which prescribes also different local actions to be taken. In combination with the fully decentralized operation, local decision making based on partial views of the whole system can introduce inconsistencies and oscillations.

As a running example, we use a simplified scenario from the above mentioned case study. It is based on the following assumptions:

(i) drivers are bound to their vehicles, i.e., there is no car sharing or car pooling possibility;
(ii) vehicles do not send parking requests to parking stations, but just use the parking stations' availability information in order to plan their trip (and re-plan if needed);
(iii) when planning, vehicles consider parking at parking stations that are within a fixed distance to the POIs in the driver's calendar.

The last point is illustrated in Fig. 1, where a vehicle follows a route that leads to two available parking stations close to its POIs (left hand-side); when one of them becomes unavailable, the vehicle has to head to the next available parking station within the radius of its first POI (right hand-side).

2.1 DEECo Model of the Running Example

The above scenario has been implemented in the DEECo component model [5]. DEECo is a component model developed within ASCENS, that targets the development and deployment of CPS following the paradigm of ACEs.

In DEECo, each component is an independent unit of development and deployment. Examples of two DEECo components in the DEECo domain specific language (DSL) are depicted in Listing 1.1. It consists of *knowledge*, i.e., component's data (e.g., lines 9-10 and 19), and one or more *processes* (e.g., lines 11-14 and 20-23). Each process is essentially a thread that operates upon the knowledge by reading the input knowledge, executing the process body and writing

```
1  role AvailabilityConsumer:
2      calendar, availabilityList
3
4  role AvailabilityProvider:
5      position, availability
6
7  component Vehicle42 features AvailabilityConsumer:
8      knowledge:
9          calendar = {(WORK,09:00,(50.846232,49.469774)),...}, availabilityList = {(23,8),... },
10         plan = {(20m,LEFT),...}, planFeasibility = TRUE, ...
11     process computePlanWhenFarFromPOI(in calendar, in availabilityList, in planFeasibility, out plan):
12             plan ← JourneyPlanner.computePlan(calendar,availabilityList, planFeasibility)
13         scheduling: periodic( 6000ms ) and triggered(changed(planFeasibility) ∨ changed(availabilityList))
14         mode: farFromPOI
15     ...
16
17 component ParkingStation23 features AvailabilityProvider:
18     knowledge:
19         position = {50.846296, 49.461009}, availability = 8, ...
20     process monitorAvailability(out availability):
21             availability ← Sensors.getCurrentAvailability()
22         scheduling: periodic( 3000ms )
23         mode: closeToPOI, farFromPOI
24     ...
25 ...
26 // updates Vehicles' belief over the availability of relevant Parking Stations
27 ensemble UpdateAvailabilityWhenFarFromPOI:
28     coordinator: AvailabilityConsumer
29     member: AvailabilityProvider
30     membership:
31         ∃ poi ∈ coordinator.calendar: distance(member.position, poi.position) ≤ THRESHOLD
32     knowledge exchange:
33         coordinator.availabilityList ← {m.availability | m ∈ members}
34     scheduling: periodic( 6000ms )
35     mode: farFromPOI
36 ...
```

Listing 1.1. Example of a DEECo component and ensemble definition in DSL.

the output knowledge. Process execution can be periodic (e.g., line 22), event-based (where an event is a change in the knowledge of the component), or both (e.g., line 13). Each process is bound to one (e.g., line 14) or more (e.g., line 23) *modes* and gets executed only if the containing component is in one of the process's modes. Finally, each component features one or more *roles* (e.g., line 7 and 17). A role is a collection of knowledge fields (e.g., lines 1-2 and 4-5).

In our running example, the two components depicted at the instance level in Listing 1.1 are `Vehicle` and `ParkingStation`. The former features the role of aggregating the parking availability information, which the later should provide. Among others, `Vehicle` comprises a process responsible for the computing the `Vehicle`'s plan, while `ParkingStation` comprises a process responsible for sensing the current availability (equivalently occupancy) of the station.

DEECo components do not have explicit bindings to each other and are not allowed to communicate directly. Instead, communication in DEECo is implicit and takes the form of *knowledge exchange* within emerging groups called *ensembles*. Forming of ensembles is one of the tasks of the DEECo runtime framework.

An ensemble in DEECo DSL is an interaction template (e.g., Listing 1.1, lines 27-35) that consists of the specification of the roles of the interacting parts, termed *coordinator* (line 28) and *member* (line 29), the specification of the condition of interaction, termed *membership* (lines 30-31), and the specification of the actual *knowledge exchange* function (lines 32-33). Similar to DEECo processes, knowledge exchange within an ensemble is triggered in a periodic (e.g., line 34) or event-triggered fashion, and is bound to the mode of the evaluating component (line 35).

In the running example, the `UpdateAvailabilityWhenFarFromPOI` ensemble specifies that whenever two component that feature the roles of `Availability-Consumer` and `AvailabilityProvider` and satisfy the condition of the latter being close to one of the POI of the former (according to their knowledge valuations), then the `availability` knowledge of the provider has to be copied to the consumer side. This models the scenario of a car that communicates with a parking station in order to obtain the station's availability and plan accordingly.

In the rest of the chapter, we will focus on the problem of *how to come up with a specification of a DEECo-based system* (such as the one depicted in Listing 1.1) based on the initial requirements and domain assumptions. Throughout the rest of the text, we illustrate the approach on the running example.

3 The Need for a Tailored Design Method for ACEs

Although DEECo provides a set of concepts (autonomous components, periodic processes, ensembles) that effectively deal with the dynamicity and distribution at a middleware level, the systematic design of CPS based on ACEs remains a significant challenge. Contemporary design methods for complex systems typically consist of the phases of (i) eliciting and analyzing the goals of the system-to-be, i.e., what is to be achieved and why, (ii) translating them into requirements specifications of the the system-to-be, and (iii) deriving the architecture of the system-to-be by mapping each requirement to one or more runtime entities (usually referred to as components). KAOS [18,19] and Tropos [3,11] are two prominent goal-oriented requirements engineering methodologies that are primarily concerned with the first two design phases. SOTA [1] and ARE [21] are two requirements modeling approaches developed within ASCENS and tailored to the domain of autonomous and self-adaptive systems that also focus exclusively on the first two design phases.

The underlying idea of KAOS is to use goals to capture the intent (the "why") behind the functionality of the system-to-be. Goals in KAOS are iteratively decomposed into sub-goals until they reach the level where they can be mapped to requirements or assumptions of the system-to-be. The process then continues with assigning each requirement to an individual system agent. Goals in KAOS can be formalized in real-time linear temporal logic (LTL) [2] and used to check a requirements specification for consistency, completeness and pertinence. Although KAOS is a well-established methodology in requirements engineering with strong focus on formal specification and reasoning, its application in the

design of ACEs is not straightforward. The main issue is that, although there have been preliminary efforts towards this [17], KAOS does not provide a smooth alignment between requirements with architecture (third design phase mentioned in the previous paragraph). For instance, if a goal in our running example is to "maintain the availability of the parking stations up-to-date", the way to reflect this goal in system architecture is open to interpretation and heavily depends on the underlying component model used for development and deployment.

Tropos is an agent-oriented methodology where goals, soft-goals, tasks and dependencies are analyzed from the perspective of the individual agents in the system-to-be. Tropos uses the i* notation [23] for producing goal and actor models, which are later mapped to agent architectures that follow the Belief-Desire-Intention (BDI) reference model [20]. In this respect, it is more effective than KAOS in aligning system requirements with system architecture and implementation. When applied to the design of ACEs, the main shortcoming of Tropos is that it fails to address the special concerns of the ACEs domain, i.e., that of distributed dynamic feedback loop-based systems. In such systems, it is important to capture the relation of the system with the environment at every time instant, as opposed to focus on future states (the case of goals in Tropos).

SOTA [1] is a requirements modeling approach for the domain of ACEs and autonomic systems in general. 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, but not to guide the requirements-driven design of ACEs. A mathematical formalization of SOTA is provided by the *General Ensemble Model* (GEM) [14] The GEM semantics of SOTA is based on timed streams of domain states which closely corresponds to the higher-order predicate semantics of IRM (cf. Section 5). Chapter II.1 [4] in this volume discusses GEM in more detail.

Autonomy Requirements Engineering (ARE) [21] is a methodology for elicitation and expression of autonomy requirements developed within ASCENS. ARE relies on goal-oriented requirements engineering approaches (such as KAOS and Tropos) to elicit and define the system goals, and uses a Generic Autonomy Requirements (GAR) model to derive and define *assistive* and eventually alternative goals (or objectives) of the system. However, similar to classical goal oriented approaches, ARE focuses on the requirements phase and not on the mapping between requirements and architecture. ARE is discussed in detail in Chapter III.3 [22] of this volume.

A key challenge in the design of ACEs is to provide a concept that, contrary to the system goal, captures the *operational normalcy* at every time instant, i.e., the property of being within certain limits that define the range of normal operation of the system. The next challenge is to use this concept in order to systematically map situation-specific high-level goals to low-level artifacts of sys-

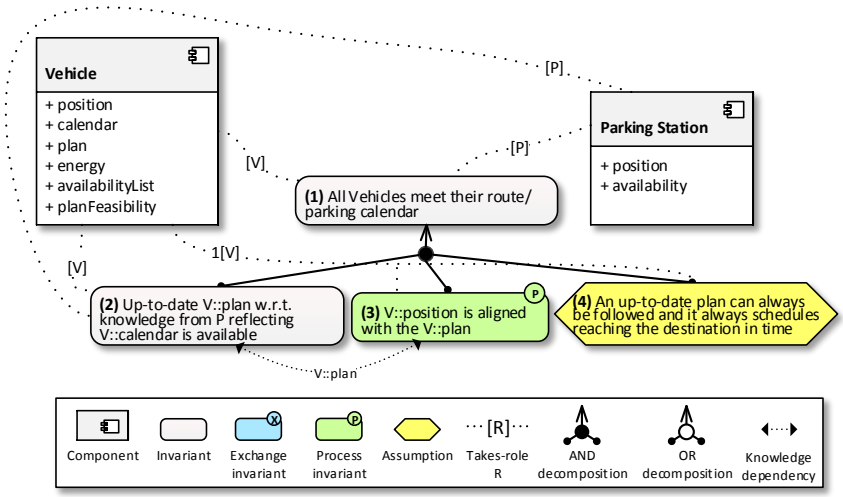

Fig. 2. Top-level design of the case study.

tem architecture (e.g., component processes, ensemble specifications, component modes) so that the compliance of design decisions with the overall system goals and requirements is explicitly captured and (if possible) formally verified.

4 Invariant Refinement Method

We have addressed the above challenge by introducing a novel design method called IRM (Invariant Refinement Method), which specifically focuses on ACEs. IRM builds on goal-based requirements elaboration as pioneered by [16]. Similar to [16], IRM focuses on the system-to-be from a global perspective and reasons about the goals and requirements of the system as whole. By gradual refinement it allows refining these goals down to the responsibilities of individual components, component processes, and ensembles.

IRM captures goals and requirements of the systems as *invariants* that describe the desired state of the system-to-be at every time instant. This corresponds to the operational normalcy of the system-to-be and thus it aligns well with the need of continuous operation of ACEs.

Fig. 2 illustrates an IRM refinement tree reflecting the running example. Each rounded rectangle represents system's requirements represented by an invariant – e.g., the top-level invariant (1): "`All Vehicles meet their route/parking calendar`".

4.1 Invariants and Assumptions

The IRM tree employs first class entities – *invariants*, *assumptions*, and *components*. A component is a primary functional entity of the system-to-be (e.g., `Vehicle` and `Parking Station` in Fig. 2). At the abstraction level of IRM, each component comprises specific knowledge, i.e., its domain-specific data. The valuation of components' knowledge evolves over time as the result of their autonomous behavior (i.e., execution of the associated component processes) and knowledge exchange. Also, a component may take up a particular role (i.e., a responsibility) in the system-to-be. This is a consequence of being referred to by an invariant.

Technically, an invariant establishes a condition over the knowledge valuation of a set of components. An invariant references components by role names – e.g., in the invariant (1) the component `Vehicle` takes the *role* `V` while `Parking Station` takes the role `P`. This way, an invariant captures the operational normalcy of the system-to-be or its logical parts (i.e., groups of components).

Invariants need not only describe the responsibilities of components, but they may also express *assumptions* about the environment. An *assumption* is thus a condition that is expected to hold during knowledge evolution and is not intended to be maintained explicitly by the system-to-be (in figures depicted as yellow hexagon; e.g., (4) in Fig. 2).

4.2 Invariants vs. Computation Activities

The core idea of IRM is that each invariant which is not an assumption is associated with a *computation activity* – an abstract computation that produces *output knowledge* given a particular *input knowledge* such that the invariant (over the input and output knowledge) is satisfied. This way, the computation activity provides a dual view on the invariant – while the invariant reflects the operational normalcy, the computation activity represents means for maintaining it.

For instance, Fig. 3 provides the dual view of computation activities reflecting the invariants in Fig. 2.

The duality of the invariants and computation activities gives a convenient option of refer to invariants for the purpose of logic-based reasoning and refer to computation activities when low-level implementation aspects are of concern.

An abstract computation activity can be related to an invariant at any level in the IRM tree. The computation activity however gets a special significance for the leaves of the IRM tree, where it corresponds to a component process or a knowledge exchange. Thus, following the dual perspective, the goal of IRM is to refine high-level invariants (i.e., the abstract activities) to the very concrete invariants which via their computation activities lead to the design of component processes and knowledge exchange.

Note that the activities associated with high-level system goals are abstract, representing the whole system implementation. At this level of abstraction, not all input knowledge can be precisely identified, this is exemplified in Fig. 3, where

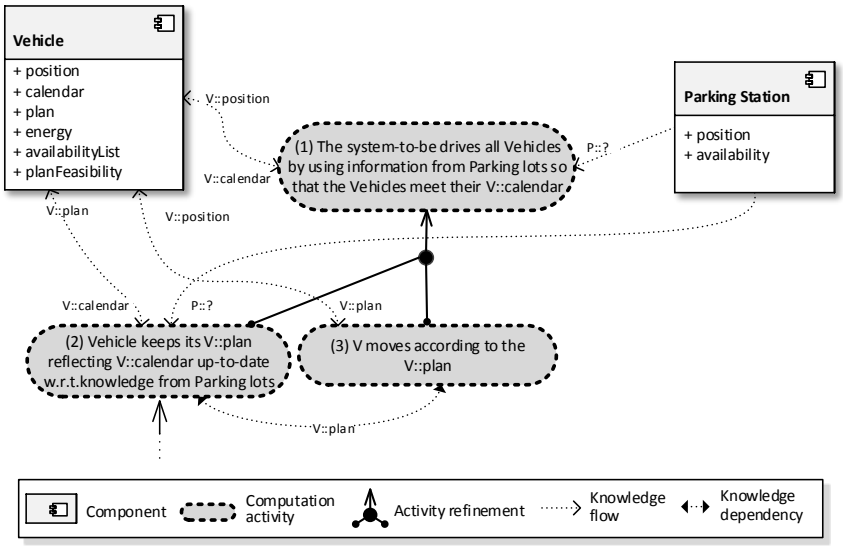

Fig. 3. Dual, activity-based view on the top-level design of the case study from Fig. 2].

the input knowledge of the activity associated with (1) comprises `V::calendar` and potentially some knowledge of parking lots, which is not yet clear, thus denoted by `P::?`. The output knowledge comprises `V::position`, which is the knowledge the evolution of which the system can effectively control by the activity.

The relations between component knowledge and input/output knowledge of activities are captured as *knowledge flows* on the IRM diagram. For example, Fig. 3 shows the knowledge flow between the `Vehicle` and the activity associated with (3) (with `V::plan`, resp., `V::position` as its input, resp., output knowledge).

4.3 Invariant Refinement

The basic process in IRM is a systematic, gradual refinement of a higher-level invariant by means of its decomposition (i.e., structural elaboration) into a conjunction or disjunction of lower-level sub-invariants, i.e., $I_p \rightsquigarrow I_{s1} \wedge \ldots \wedge I_{sn}$ and $I_p \rightsquigarrow I_{s1} \vee \ldots \vee I_{sn}$.

Formally, decomposition of a parent invariant I_p into a conjunction of sub-invariants I_{s1},...,I_{sn} is a refinement if the conjunction of the sub-invariants entails the parent invariant, i.e., if it holds that:

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

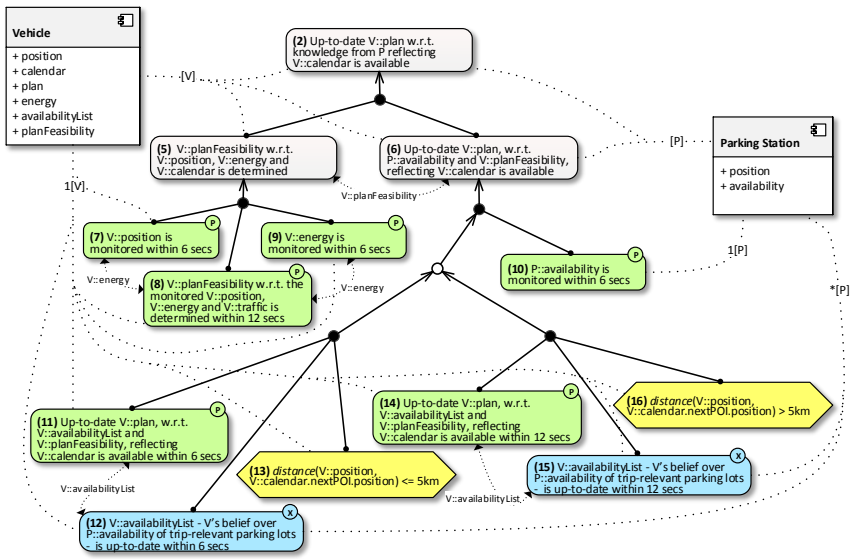

Fig. 4. Invariant refinement of "`Up-to-date V::plan w.r.t. knowledge from P reflecting V::calendar is available`".

This definition follows the classical interpretation of refinement, where the composition of the children exhibits all the behaviors expected from the parent and (potentially) some more.

Similarly, the OR-decomposition of a parent invariant I_p into the sub-invariants I_{s1},...,I_{sn} is a refinement if it holds that:

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

The refinement via AND- and OR-branching is applied recursively. This starts with high-level invariants that reflect the overall system goals and involve a number of components and ends with low-level invariants that involve a single component or an ensemble of components.

To keep the semantics of the refinement, only the components that take a role in the parent invariant may also take a role in the sub-invariants. However, as the refinement also leads to concretization of the problem and its solutions, new knowledge can be added to the components that take a role in the sub-invariants (e.g., `V::planFeasibility` in Fig. 4).

The process of refinement is demonstrated in Fig. 2. As a design decision the invariant (1) is refined 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 refinement further continues in Fig. 4, where the invariant (2) is further refined, up to the leaves.

Note the OR-decomposition below the invariant (6). Formally, the IRM refinement allows only AND-refinement or OR-refinement, but not a combination of both. If a combination is necessary, it has to be formally modeled by introduction of synthetic invariants following the abstract-syntax tree of the desired formula. As these synthetic invariants do not provide any additional knowledge, the graphical notation used in the figure omits them and permits direct connection of refinement symbols.

Seeing the refinement from the dual perspective of computation activities, the computation activity of a parent is formed by parallel execution of its sub-activities. In case of AND-refinement, this involves all sub-activities. In case of OR-refinement, this involves executing exactly one sub-activity. To help determine which sub-activity of an OR-refinement to execute, the design practice is to equip each OR-branch with an assumption which acts as a guard to the branch (see assumptions (13) and (16) in Fig. 4).

4.4 Leaves of Refinement

As the rule of thumb the refinement is finished when each leaf invariant of the refinement tree is either an assumption or is a computation activity corresponding to a *process* or *knowledge exchange* (see Section 2.1). In particular, the invariant corresponds to a process if it captures the operational normalcy of a single component (technically it means that it refers only to knowledge of a single component). Such an invariant is called a *process invariant* (in diagrams marked by `P`, e.g., (3) in Fig. 2).

Similarly, an invariant corresponds to knowledge exchange (called *exchange invariant*) if it captures the fact that the knowledge of one component is in certain relationship (typically in "identity" relationship) to knowledge of another component. Invariant (12) in Fig. 4 is in example of an exchange invariant. Exchange invariants are marked by `X`.

Generally, it is possible to refine invariants where several components take a role (e.g., (5)) to process and exchange invariants which are eventually associated with "real" computation activities. This typically involves a number of refinement steps in which (a) the invariants are gradually split by roles and (b) exchange invariants are introduced that collect needed knowledge.

Specifically, to refine an invariant I_p, referencing the components C_1,...,C_m into sub-invariants I_{s1},...,I_{sn} we introduce the *belief of* C_1 *over the knowledge of* C_2,...,C_m. In this context, the belief $B_{C_1}^{C_2,\ldots,C_m}(K)$ is knowledge of C_1 that represents C_1's snapshot of a part K of the knowledge of C_2,...,C_m. For instance, in Fig. 4, the belief `V::availabilityList` of `Vehicle` over the knowledge `P::availability` of `Parking Stations` is an example of such a knowledge snapshot (denoted as `V::availabilityList`=$B_{Vehicle}^{ParkingStation}$(`P::availability`)).

Thus, I_{s1} formulates the normalcy properties of $B_{C_1}^{C_2,...,C_m}$, whereas I_{s2},...,I_{sn} refine I_p while substituting the references to the knowledge of C_2,...,C_m by references to $B_{C_1}^{C_2,...,C_m}$. Note that $B_{C_1}^{C_2,...,C_m}$ is a new knowledge introduced into C_1. For example, (15) formulates the condition on creating the belief `V::availabilityList`=$B_{Vehicle}^{ParkingStation}$(`P::availability`), whereas (14) refines (6) while substituting the references to `P::availability` by references to `V::availabilityList`.

Furthermore, I_{s2},...,I_{sn} are potentially process/exchange invariants, since, in general, the number of components taking a role in I_{s2},...,I_{sn} is, compared to I_p, decreased at least by one due to references to the belief $B_{C_1}^{C_2,...,C_m}$ (such as when comparing (6) and (14)).

4.5 From Invariants to Final Architecture

Once the refinement reaches the level of process and exchange invariants, the design continues to the implementation level by refining each process invariant into a component process and each exchange invariant into an ensemble. For example, in Listing 1.1 `Vehicle` is reified by `Vehicle42`, while (14) is refined into the `Vehicle42`'s `computePlanWhenFarFromPOI` process, and (15) is refined into the `UpdateAvailabilityWhenFarFromPOI` ensemble. Thus, determined by the invariant refinement, this step yields the final architecture of the system. The details are beyond the scope of this text; we refer the interested reader to [8].

5 IRM Abstraction Levels and Invariant Patterns

There is a significant abstraction gap between the high-level and low-level invariants. The former ones capture general operational normalcy while the latter ones reflect architectural elements and thus capture the ACEs-specific aspects. In this section we provide a detailed description of bridging the gap during the invariant refinement, i.e., during generation of low-level invariants from high-level ones. We have identified five patterns of invariants that reflect the way operational normalcy is captured at four adjacent abstraction levels that are covering the abstraction gap. With these patterns we can precisely set out the rules and guidelines for refinement of the invariants on the same/adjacent abstraction levels. The rules/guidelines allow for iterative refinement to continuously lower the level of abstraction until the architectural elements level is reached.

In particular, the patterns are as follows (from the most abstract to the least abstract):

1. *general invariants*,
2. *present-past invariants*,
3. *activity invariants*,
4. *process invariants*, and
5. *exchange invariants* (patterns 4. and 5. are at the same level of abstraction).

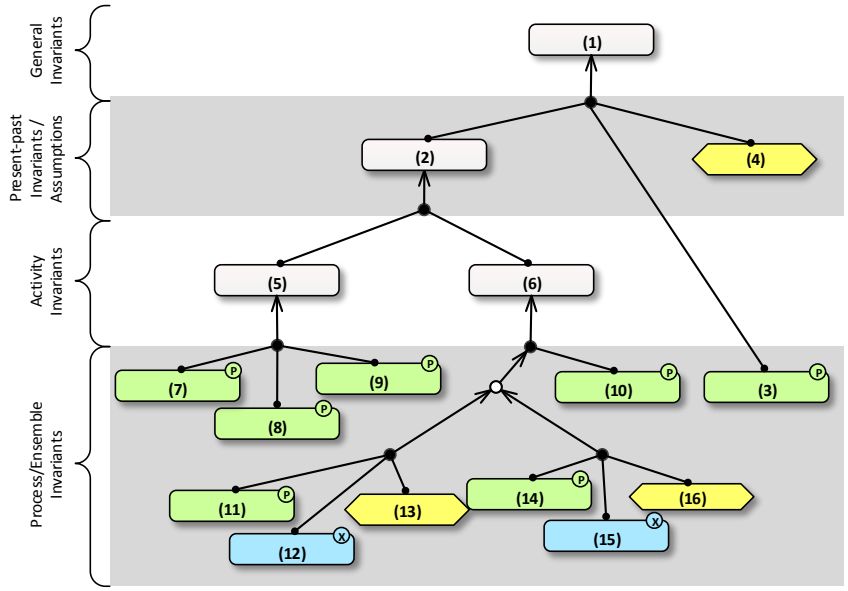

Fig. 5. Patterns of invariants in the case study.

The patterns of invariants in the case study are illustrated in Fig. 5.

In the rest of the section we use a predicate formalization of the invariants in order to allow their precise definition and, in particular, to highlight the differences between the patterns. In principle, an invariant expresses the operational normalcy in terms of a condition to be maintained during knowledge evolution in time. Using this formalization, it is possible to refer to timed sequences of the knowledge values, and thus it allows for viewing the complete knowledge value evolution in time. An important aspect of ACEs-based systems is that they are inherently asynchronous. Thus, the formalization has to capture the evolution in terms of asynchrony and delays. As an example, the knowledge evolution shown in Fig. 6 can be assumed. In it, we are interested in a formalization of the form "*The value of V::pAvailable always equals the value of P::available that is not older than the period*" rather than in "*V::pAvailable equals P::available*" (which does not always hold).

5.1 Formalization

We formalize the invariants as follows.

Definition 1. *Time is represented by a non-negative real number, i.e.,* $\mathbb{T} \stackrel{def}{=} \mathbb{R}_0^+$.

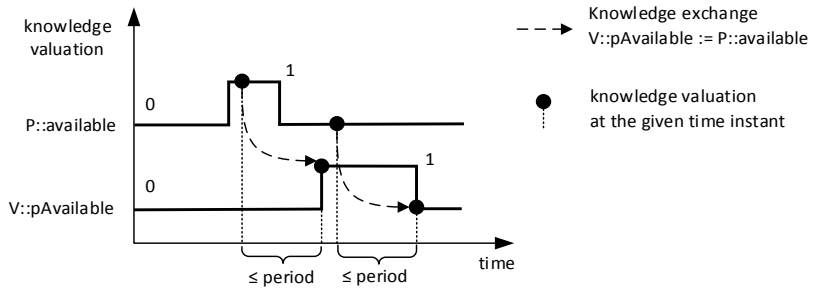

Fig. 6. Example of knowledge evolution in time when employing (periodic) knowledge exchange.

Definition 2. *Knowledge is a set $K = k_1, ..., k_n$ of knowledge elements, where the domain of k_i is denoted as V_i.*

Definition 3. *Knowledge valuation of element k_i is a function $T \to V_i$ which for a time t yields a value of k_i (denoted as $k_i[t]$).*

Definition 4. *Invariant is a predicate (in a higher-order predicate logic with arithmetic) over knowledge valuations and time.*

Of course, the invariant definition above is not the only possible one. For example, real-time LTL [2] can be used too. Nevertheless, we use the proposed formalization as our primary goal is not model checking but rather a description of invariant refinement. For it, we believe the formalization is more suitable and allows for straightforward formulating and proving relevant theorems.

In the rest of this section, we detail the identified invariant patterns and provide formal definitions as well as macros to ease their usage.

5.2 General Invariants

General invariants are defined at the top-level of abstraction and they capture the operational normalcy by relating the past and current knowledge valuations to a future knowledge valuation.

An example of this pattern is the invariant (1): "*All Vehicles meet their route/parking calendar*", which can be formalized as follows (for the sake of brevity, it assumes only the calendar with a single POI not changing in time):

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

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

5.3 Present-past Invariants

On the lower level of abstraction, there are *present-past* invariants that capture the operational normalcy employing the current and/or past knowledge valuations. This corresponds with the fact that software systems can work with current and/or past data and cannot depend on future data. This fact has been abstracted away at the level of general invariants. To limit the amount of needed past data, the *lag* of a present-past invariant is defined as the maximal distance in the past that is needed to formulate the operational normalcy of the invariant. As in real-time software control systems, it is assumed that the smaller the lag, the bigger precision and robustness and vice-versa. An idealized and unreachable case is the zero lag, which would mean that the beliefs of all components are always up-to-date and their actions are instant.

Importantly, when a general invariant is refined into present-past invariants (or more precisely into a conjunction of them), assumptions have to be added that guarantee that maintaining the operational normalcy based on the current/past knowledge valuation will eventually result in reaching the operational normalcy based on a future knowledge valuation. An example of such an assumption is the assumption (4) in Fig. 2.

Definition 5. *(Present-past invariants) For a predicate P capturing the relation between valuation of knowledge elements $I_1, \ldots, I_n$ and $O_1, \ldots, O_m$, and the lag L, the expression $P^L_{p-p}[I_1, \ldots, I_n][O_1, \ldots, O_m]$ denotes the following* present-past invariant*:*

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

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

Invariant (2): "*Up-to-date V::plan, w.r.t. knowledge from P, reflecting V::calendar is available*" is an example of a present-past invariant. For parking lots $P_1..P_n$ and lag L it can be formalized as follows:

"*At any time, for the current valuation of* `V::plan` *there is a valuation of knowledge of P_1, ..., 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:

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

In this predicate, if the lag L greater than zero, it means that the `V::plan` is outdated regarding the current knowledge of the parking lots (the greater $L \Rightarrow$ more outdated parking-lot knowledge valuation). The zero lag would mean the plan is up-to-date at any moment.

Using the shortcut introduced in Definition 5, we can rewrite the expression as:

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

Such an shortcut can be also used 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.

5.4 Activity Invariants

Frequently, properties of a (soft) real-time activity have to be assumed. A commonly used property is then that each output knowledge valuation is based on the same or newer input knowledge valuation than the previous one. Fig. 7 illustrates this.

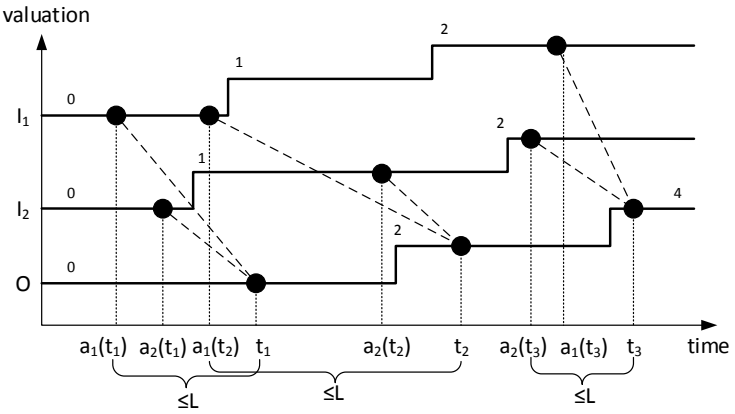

Fig. 7. Illustration of a valid knowledge valuation with respect to an activity where the output O represents the sum of inputs I_1 and I_2, while meeting lag L.

An *activity invariant* expresses the fact that the output knowledge valuation changes only as a result of performing the activity. Moreover, the activity never exceeds the corresponding time limit (the lag). More rigorously, at any time the output knowledge valuation corresponds to the outcome of the activity applied on input knowledge valuation not older than the lag. Plus, each output is based on the same or newer inputs than the previous output.

Definition 6. *(Activity invariant) For a predicate P reflecting the post-condition of an activity with inputs $I_1, \ldots, I_n$ and outputs $O_1, \ldots, O_m$, and for lag L, the expression $P^{L}_{act}[I_1, \ldots, I_n][O_1, \ldots, O_m]$ denotes the following* activity invariant*:*

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

where the non-decreasing function a_i gives for each time t the corresponding time t' such that the valuation of I_i at t' was "used to compute" the valuation of $O_1, \ldots, O_m$ at t, as shown in Fig. 7.

Invariant (6) can serve as an example of the activity invariant: "*Up-to-date V::plan, w.r.t. P::availability and V::planFeasibility, reflecting V::calendar is available*". For parking lots $P_1..P_n$ and lag L it can be 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.*"

Using the predicate logic, it can be expressed as follows:

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

The aspect that `V::plan` may change only as the result of an execution of a planning activity is captured by the usage of the non-decreasing function $a_i : \mathbb{T} \to \mathbb{T}$ rather than a particular $t_i \in \mathbb{T}$. The a_i function also captures the read consistency.

Similarly as in the previous invariant, the lag greater than zero means that the outdated valuation of `P::availability` and `V::planFeasibility` is considered. The zero lag 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 (i.e., the activity computes infinitely fast and infinitely often).

Using the shortcut introduced in Definition 6, we can write the formalization of invariant (6) as:

$$UpToDatePlan^{L}_{act} \left[\begin{array}{c} P_1{::}availability[a_n(t)], \\ \vdots \\ P_n{::}availability[a_n(t)], \\ V{::}planFeasibility[a_{pF}(t)], \\ V{::}calendar[a_{cal}(t)], \\ V{::}plan[t] \end{array} \right] \left[V{::}plan \right]$$

5.5 Process Invariants

Process invariants are in the leaves of invariant decomposition, i.e., at the lowest level of abstraction. Such an invariant captures a periodic real-time component process. Into it, an activity invariant capturing local computation (while assuming read consistency) is refined.

Contrary to the activity invariants, the process invariant adds a constraint that the activity is executed exactly once in every *period*. Therefore, the period can be seen as a refinement of the activity lag and the output knowledge evaluation is determined by the release time (time at which a task becomes ready for execution) and finish time in each period [9].

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

Definition 7. *(Process invariant) For a predicate P reflecting the post-condition of a periodic real-time process with inputs $I_1, \ldots, I_n$, outputs $O_1, \ldots, O_m$, and period L, the expression $P^L_{proc}[I_1, \ldots, I_n][O_1, \ldots, O_m]$ denotes the following* process invariant*:*

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

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

In contrast to the activity invariant, there is the same R for every input I. It reflects the fact that at the release time, all the inputs are read by the process atomically.

The invariant (11) can be taken as an example of the process invariant: "*Up-to-date V::plan, w.r.t. V::availabilityList and V::planFeasibility, reflecting V::calendar is available*". For period L, it can be 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:

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

$$t \geq F(p) \Rightarrow UpToDatePlan \begin{pmatrix} V{::}availabilityList[R(p)], \\ V{::}planFeasibility[R(p)], \\ V{::}calendar[R(p)], \\ V{::}plan[t] \end{pmatrix}$$

where $E : \mathbb{N}_0 \to \mathbb{T}$ and $E(n) = n \cdot L$, i.e., the end of the n-th period. $R(n)$ and $F(n)$ denote the release and finish time of the real-time process in the n-th period, as per Definition 7.

In the process invariant case, the zero L means that the `V::plan` is at each time instant infinitely close to the up-to-date plan with respect to the current `V::availability`, `V::planFeasibility`, and `V::calendar` of the vehicle.

With the help of the shortcut from Definition 7, the formalization of (11) can be shortened as:

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

5.6 Exchange Invariants

The activity invariants that capture the establishment of a belief (that can be addressed by ensemble knowledge exchange) while assuming distributed read consistency, are refined into *exchange invariants*, which capture periodic knowledge exchange of an ensemble.

In contrast to process invariants, the input values in exchange invariants can be obtained at different times (but the times still have to belong to the same period), as the input values are potentially distributed. Additionally, knowledge propagation delays are also considered. These delays can arise for example from delays in network communication.

In summary, the exchange invariants depict a composite activity composed of knowledge transfer and periodic evaluation of the knowledge exchange and membership condition.

Importantly, each component processes the incoming knowledge exchange by itself. The required input knowledge is sent asynchronously by other components. If the knowledge transfer time is larger than the knowledge exchange period, the composite activities may partially overlap.

Definition 8. *(Exchange invariant) Let P be a predicate reflecting the post-condition of a periodic knowledge exchange with inputs $I_1, \ldots, I_n$, outputs $O_1, \ldots,$*

O_m, and period L. Provided that it takes at most T for the knowledge to become available at the component executing the knowledge exchange, the expression $P_{exc}^{L,T}[I_1,\ldots,I_n][O_1,\ldots,O_m]$ denotes the following exchange invariant*:*

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

where $E:\mathbb{N}_0\to\mathbb{T}$ and $E(n)=n\cdot L$, i.e., the end of the n-th period. $R(n)$ and $F(n)$ denote the release and finish time of the real-time knowledge exchange in the n-th period. Finally, a_i gives for each time t the corresponding time t' such that the valuation of I_i that was available to the component executing the knowledge exchange at t was sent to the component at t'.

For every input I_i, the a_i can be a different value as the component executing the knowledge exchange can receive the inputs at different times. On the other hand, the knowledge exchange is assumed to be unidirectional. It means that that the exchange is written into the knowledge of a single component only, and therefore these writes can be atomic. Thus, for every output O_i there is the same t.

The invariant (12) of the running example can taken as a representative of the exchange invariant: "*V::availabilityList – Vs belief over P::availability of trip-relevant parking lots – is up-to-date*". For parking lots $P_1..P_n$, period L, and upper bound for knowledge transfer T it can be 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.*"

The predicate logic can capture it as follows:

$$\exists a_1,...,a_n : \mathbb{T}\to\mathbb{T}, 0<x-a_i(x)\le T, \forall i\in\{1..n\},$$
$$\exists R,F:\mathbb{N}\to\mathbb{T}, E(x-1)\le R(x)<F(x)<E(x),$$
$$\forall p\in\mathbb{N}, \forall t\in\langle E(p-1),E(p)\rangle :$$
$$t<F_V(p)\Rightarrow EqualsRelevant\begin{pmatrix} P_1{::}availability[a_1(R(p-1))],\\ \vdots\\ P_n{::}availability[a_n(R(p-1))],\\ V{::}availabilityList[t]\end{pmatrix}$$
$$t\ge F_V(p)\Rightarrow EqualsRelevant\begin{pmatrix} P_1{::}availability[a_1(R(p))],\\ \vdots\\ P_n{::}availability[a_n(R(p))],\\ V{::}availabilityList[t]\end{pmatrix}$$

In this case, zero L means that, at each time instant, the `V::availabilityList` is infinitely close to the set of the current `P::availability` of all the relevant parking lots.

With the help of the shortcut from Definition 8, the invariant (12) can be formalized as:

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

5.7 Refinement Between Invariant Patterns

With the invariant patterns described, we can now introduce the guidelines for decomposition at the corresponding levels of abstraction. The goal of these guidelines is to guarantee the refinement between invariants following the patterns. The guidelines are presented here informally only; the formal definitions and proofs are in [7].

General → Present-past. As already mentioned in section 5.3, when a general invariant is refined into (a conjunction of) present-past invariants, assumption invariants have to be introduced (e.g., invariant (4) in Fig. 2). From the formal point of view, this refinement is the most demanding one as it is necessary to proof each case separately.

Present-past → Present-past. When a single present-past invariant is refined into a conjunction of other present-past invariants, the combined lag of the sub-invariants is not greater that the parent's lag. The combination is figured out by the knowledge dependencies among the sub-invariants. (By knowledge dependency, we mean here a situation, when an invariant uses knowledge produced by the activity associated with another invariant.)

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. As in the case of present-past → present-past invariant refinement, an activity invariant can also refined into a conjunction of other activity invariants. 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 [9].

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

Impact of IRM Design on the Case Study. By identifying invariants in the case study, classifying them into the available invariant patterns (Sections 5.2-5.6), and subsequently refining them using the above guidelines, we systematically constructed the IRM tree of the case study. This can be used in turn to derive the DEECo specification of the case study (Listing 1.1 in Section 2.1).

6 Conclusions

In this chapter, we have presented IRM – a requirements elicitation and architectural design method that guides the design of ACEs. With respect to the Ensemble Development Life Cycle (cf. Chapter III.1 [13]), IRM lies in the transition between the Requirements Engineering and the Modeling Phase of the design wheel. IRM takes similar approach as found in goal-oriented requirements engineering, but specifically focuses on "maintain" goals, as these are critical for continuously running systems that constantly interact and control their environment (such as cyber-physical systems).

The core idea of IRM is to describe the variability of a system by AND and OR invariant decompositions that capture the required functionality of the system under different runtime situations. IRM establishes systematic refinement between high-level requirements and low-level architectural concepts, i.e., components, component processes, and knowledge exchange functions as defined in the DEECo component model. This directly allows deriving an architecture of ACEs and brings about strong traceability.

References

1. Abeywickrama, D.B., Bicocchi, N., Zambonelli, F.: SOTA: Towards a General Model for Self-Adaptive Systems. In: Proc. of WETICE '12. pp. 48–53. IEEE (2012)
2. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of Real-Time Properties. In: Proc. of FSTTCS '06. pp. 260–272. Springer (2006)
3. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An Agent-Oriented Software Development Methodology. Autonomous Agents and Multi-Agent Systems 8(3), 203–236 (May 2004)
4. Bruni, R., Corradini, A., Gadducci, F., Hölzl, M., Lafuente, A.L., Vandin, A., Wirsing, M.: Reconciling White-Box and Black-Box Perspectives on Behavioural

Self-Adaptation. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, Heidelberg (2015)
5. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: DEECo – an Ensemble-Based Component System. In: Proc. of CBSE'13. pp. 81–90. ACM (2013)
6. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F., Plouzeau, N.: Adaptation in Cyber-Physical Systems: from System Goals to Architecture Configurations. Tech. rep., D3S-TR-2014-01, Dep. of Distributed and Dependable Systems, Charles University in Prague (Jan 2014), `http://d3s.mff.cuni.cz/publications/download/D3S-TR-2014-01.pdf`
7. Bures, T., Gerostathopoulos, I., Keznikl, J., Plasil, F., Tuma, P.: Formalization of Invariant Patterns for the Invariant Refinement Method. To appear in Springer LNCS volume dedicated to Wirsing-Festschrift, preliminary version available at `http://d3s.mff.cuni.cz/publications/download/D3S-TR-2013-04.pdf` (2015)
8. Bures, T., Gerostathopoulos, I., Vojtech, H., Keznikl, J., Kofron, J., Loreti, M., Plasil, F.: Language Extensions for Implementation-Level Conformance Checking, ASCENS deliverable 1.5 (Nov 2012), `http://www.ascens-ist.eu/deliverables`
9. Buttazo, G., Lipari, G., Abeni, L., Caccamo, M.: Soft Real-Time Systems: Predictability vs Efficiency. In Computer Science, Springer (2005)
10. Gerostathopoulos, I., Bures, T., Hnetynka, P.: Position Paper: Towards a Requirements-Driven Design of Ensemble-Based Component Systems. In: Proc. of HotTopiCS workshop at ICPE '13. pp. 79–86. ACM (2013)
11. Giorgini, P., Kolp, M., Mylopoulos, J., Pistore, M.: The Tropos Methodology: An Overview. In: Methodologies and Software Engineering for Agent Systems, pp. 89–106. Kluwer Academic Publishers (2004)
12. Hoch, N., Bensler, H.P., Abeywickrama, D., Bures, T., Montanari, U.: The E-Mobility Case-Study. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, Heidelberg (2015)
13. Hölzl, M., Koch, N., Puviani, M., Wirsing, M., Zambonelli, F.: The Ensemble Development Life Cycle and Best Practises for Collective Autonomic Systems. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, Heidelberg (2015)
14. Hölzl, M.M., Wirsing, M.: Towards a System Model for Ensembles. In: Agha, G., Meseguer, J., Danvy, O. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 241–261. Springer (2011)
15. Keznikl, J., Bures, T., Plasil, F., Gerostathopoulos, I., Hnetynka, P., Hoch, N.: Design of Ensemble-Based Component Systems by Invariant Refinement. In: Proc. of CBSE '13. pp. 91–100. ACM (2013)
16. Lamsweerde, A.V.: Goal-Oriented Requirements Engineering: A Guided Tour. In: Proc. of RE'01. pp. 249–262. IEEE (2001)
17. Lamsweerde, A.V.: From System Goals to Software Architecture. In: Formal Methods for Software Architectures. LNCS, vol. 2804, pp. 25–43 (2003)
18. Lamsweerde, A.V.: Requirements Engineering: From Craft to Discipline. In: Proc. of SIGSOFT '08/FSE-16. pp. 238–249. ACM (2008)
19. Lamsweerde, A.V.: Requirements Engineering: From System Goals to UML Models to Software Specifications. John Wiley and Sons (2009)

20. Rao, A., Georgeff, M.P.: BDI Agents: From Theory to Practice. In: Proc. of ICMAS '95. pp. 312–319 (1995)
21. Vassev, E., Hinchey, M.: Autonomy Requirements Engineering. IEEE Computer 46(8), 82–84 (2013)
22. Vassev, E., Hinchey, M.: Engineering Requirements for Autonomy Features. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, Heidelberg (2015)
23. Yu, E.: Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering. pp. 226–235. IEEE Comput. Soc. Press (1997)