Formalization of Invariant Patterns for the Invariant Refinement Method

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

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

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

Keywords: architecture refinement, requirements, assume-guarantee

1 Introduction

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

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

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

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

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

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

1.1 Running Example

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

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

2 Background

2.1 Invariant refinement

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

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

$$\begin{array}{ll} I_{s1} \wedge \ldots \wedge I_{sn} \Rightarrow I_p & (\mathit{entailment}) \\ I_{s1} \wedge \ldots \wedge I_{sn} \not\Rightarrow \mathit{false} & (\mathit{consistency}) \end{array}$$

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

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

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

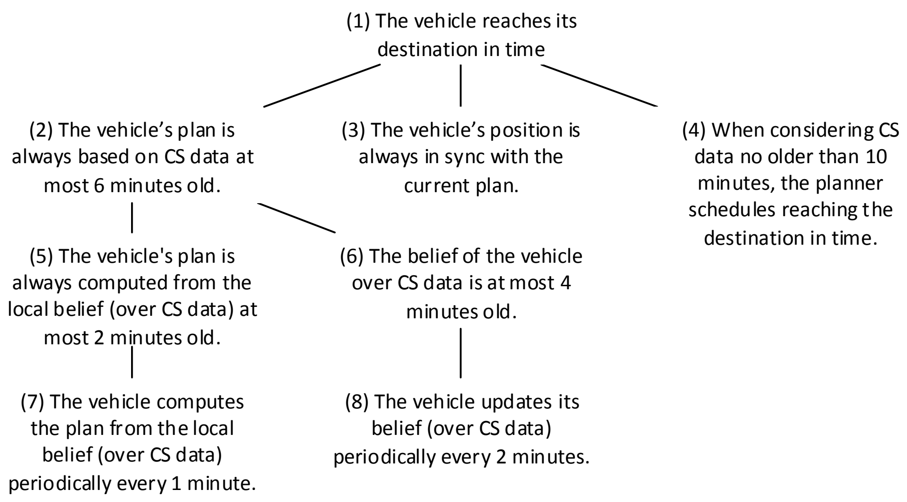
(1) The vehicle reaches its destination in time
(2) The vehicle's plan is always based on CS data at most 6 minutes old.
(3) The vehicle's position is always in sync with the current plan.
(4) When considering CS data no older than 10 minutes, the planner schedules reaching the destination in time.
(5) The vehicle's plan is always computed from the local belief (over CS data) at most 2 minutes old.
(6) The belief of the vehicle over CS data is at most 4 minutes old.
(7) The vehicle computes the plan from the local belief (over CS data) periodically every 1 minute.
(8) The vehicle updates its belief (over CS data) periodically every 2 minutes.

Fig. 1. Invariant refinement of the running example.

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

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

2.2 Invariant formalization

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

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

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

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

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

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

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

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

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

3 Invariant patterns

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

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

Present-past invariants. At a lower abstraction level, the invariants express that some knowledge is based on other knowledge, which, at the same time, is no older than a particular time interval – *lag*. This reflects the fact (abstracted by general invariants) that software systems cannot employ future knowledge to maintain their operational normalcy, but have to depend on present and/or past knowledge instead.

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

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

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

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

During refinement of a general invariant into (a conjunction of) present-past invariants, it is necessary to introduce assumptions to guarantee that maintaining the operational normalcy based on the current and/or past knowledge valuation will eventually result in reaching the operational normalcy based on a future knowledge valuation – e.g. assumption (4) in Figure 1.

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

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

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

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

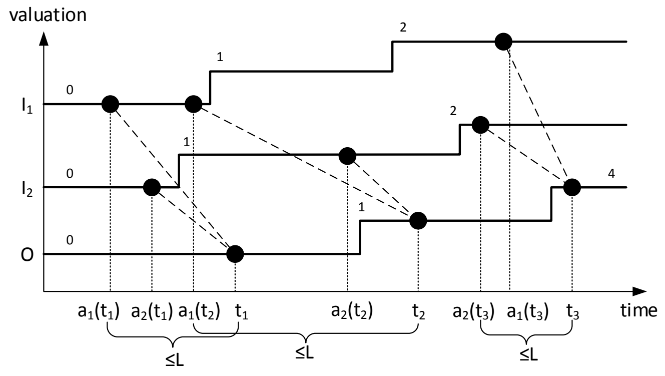

Fig. 2. Illustration of a valid knowledge valuation with respect to an activity where the output O represents sum of inputs I_1 and I_2, while meeting lag L.

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

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

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 Figure 2.

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

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

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

Definition 5. *(Process invariant) For a predicate P reflecting the post-condition of a periodic real-time process with inputs $I_1, \ldots, I_n$, outputs $O_1, \ldots, O_m$, and* *period L, the expression* $P^{L}_{proc}[I_1, \ldots, I_n][O_1, \ldots, O_m]$ *denotes the following* process invariant*:*

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

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

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

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

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

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

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

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

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

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

3.1 Illustration of invariant patterns on the running example

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

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

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

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

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

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

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

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

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

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

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

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

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

$$Belief^{4min}_{p-p}[CS_1, \dots, CS_n][v.belief]$$

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

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

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

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

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

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

4 Correctness by construction

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

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

4.1 Basic pattern relations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Similarly, it holds that the exchange invariant pattern is a refinement of the activity invariant pattern with lag equal twice the period of the exchange invariant pattern plus the time for distributed transfer of the knowledge, as formulated by the following theorem.

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

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

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

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

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

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

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

4.2 Pipeline decomposition

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$ {P_1}_{act}^{L_1} \wedge \cdots \wedge {P_p}_{act}^{L_p} \Rightarrow (P_1 \wedge \cdots \wedge P_p)_{act}^{L} $$

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

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

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

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

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

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

The assumption of the above lemma (i.e., ${P_1}_{act}^{L_1} \wedge \cdots \wedge {P_p}_{act}^{L_p}$) and the properties of the dependency chain $\mathcal{C} = \left({P_{i_1}}_{act}^{L_{i_1}}, \ldots, {P_{i_q}}_{act}^{L_{i_q}} \right)$ as a part of the pipeline decomposition $\mathcal{D}$ give us the following corollary:

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

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

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

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

By combining all the above facts, we get: ${P_1}_{act}^{L_1} \wedge \cdots \wedge {P_p}_{act}^{L_p} \Rightarrow Q_{act}^{L}$

□

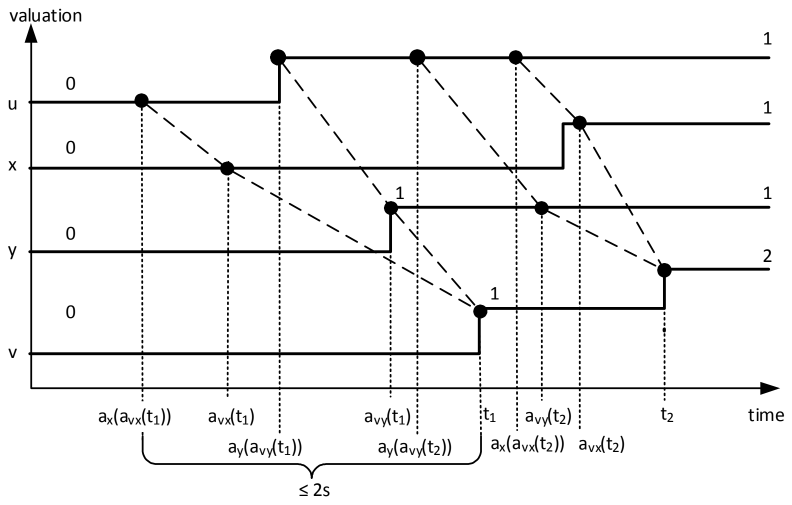

Fig. 3. A counterexample illustrating the importance of the pipeline refinement assumption in Theorem 5.

4.3 More complex types of refinement

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

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

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

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

The reason why the relaxed Theorem 5 does not work for the counterexample is that while the parent works with the valuation of a at a single time instant, the decomposition employs the valuation of a at two different time instants (by aliasing to x and y). This observation applies in general. Moreover, for some decompositions it appears that it is not possible to formulate similar theorems.

5 Discussion and Conclusions

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

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

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

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

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

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

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

References

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