Scalable Redundancy Detection for Real-Time Requirements

Elisabeth Henkel
University of Freiburg
Freiburg, Germany
0000-0003-3844-8292

Nico Hauff
University of Freiburg
Freiburg, Germany
0000-0002-8972-2776

Lena Funk
University of Freiburg
Freiburg, Germany
0009-0008-0216-0776

Vincent Langenfeld
University of Freiburg
Freiburg, Germany
0000-0001-9835-6790

Andreas Podelski
University of Freiburg
Freiburg, Germany
0000-0003-2540-9489

***Abstract*—Describing a system in a requirements specification demands correctness and conciseness. Requirements are *redundant* if they are stated multiple times throughout a specification (explicitly or implicitly). In contrast to vacuity, redundancies do not inherently indicate specification defects, and are sometimes even inevitable to adequately follow safety practices. However, intended redundancies have to be managed to avoid subsequent errors. Unintended redundancies often hint to defects in the requirements specification.**

We present an analysis for redundancies in formal real-time requirements specifications based on automata theoretical model checking. To enable this analysis, we introduce a determinism preserving totalization and complement procedure for the timed automaton model of Phase Event Automata. We state the redundancy check for a set of real-time requirements as a program analysis task.

Benchmarks show the viability of our approach to analyse requirements sets of industrial size and complexity: the analysis scales well on industrial sets, interesting redundancies both from requirements and as a formalisation artefact were found.

***Index Terms*—Formal Requirements Analysis, Real-time Requirements, Redundancy, Vacuity, Phase Event Automata**

I. Introduction

A requirements specification should describe a system that is to be built correctly, completely, and concisely. Requirements that are stated multiple times within a system specification are redundant. This may be because the requirement is stated multiple times identically, or because the conjunction of a set of requirements is already implied by the redundant requirement. While redundancy in itself is not necessarily bad, to some extent it may even increase readability of a requirements document, redundancies have to be managed in order to prevent defects during changes [1], [2]. Nonetheless, redundant requirements often hint to defects in the requirements. Redundancy in a requirements set may be caused by several defects:

A redundancy can be caused by a requirements precondition never being satisfied, caused by the sheer size of the system that is specified. This kind of defect, often called *vacuity*, can be detected by several requirements analysis tools [3], [4] and has been shown to be found in industry requirements [3]. This kind of redundancy is generally seen as a defect as vacuous requirements serve no function.

A redundancy can also be caused by a requirement being less restrictive than the composition of the remainder of the specification. This kind of redundancy may be intentional, e.g., for regulatory reasons, or due to including requirements from different stakeholders who want to know their specification to be covered. It might also stem from the intentional inclusion of requirements that prioritise over nuances of desirable system properties. The redundancy may also be non-intentional, i.e., hint to an oversight in the design or simple human error during documentation or even formalisation. Nonetheless, the redundancy has to be known, either to be managed during changes or for the requirements to be repaired. Analysis of requirements defects in industry projects has shown that the introduction of redundancies can be as simple as copy-and-paste errors and confusion of system variables can lead to a requirement specification being incomplete but redundant [5].

Formal requirements analysis tools analysing different properties such as consistency and vacuity have shown their usefulness in practical applications [6]–[9]. To our knowledge, none of these tools are able to detect arbitrary redundancy (i.e., more than vacuity, semantic equivalence or per-requirement equivalence) for formal temporal requirements.

In this paper, we present our scalable redundancy analysis for formal requirements in HanforPL. To enable this analysis, we introduce a determinism preserving totalization and complement procedure for the timed automaton model of Phase Event Automata. Further, we state the redundancy check for a set of real-time requirements as a program analysis task.

A. Motivating example

In the following, we give a motivational example of two requirements where one is subsumed by the other requirement. The following set of requirements is given in the SPL-like [3] requirements language HanforPL [10], [11]:

r_0: *If* y *holds, then* $x \geq 5$ *holds after at most* 3 *time units.*
r_1: *If* y *holds, then* $x \geq 5$ *holds after at most* 5 *time units.*

One can clearly see that requirement r_0 is a stronger restriction on the state space of the system than requirement r_1. This is, the requirements are equivalent except for a weaker time bound in r_1. It will therefore never have any influence on the state space of the system. Note that, although our example only regards subsumption because of time bounds, our approach is not limited to timing and detects any subsumption, e.g., due to constraints on observables or triviality of a requirement. Also, our approach is not limited to a pair of requirements, but

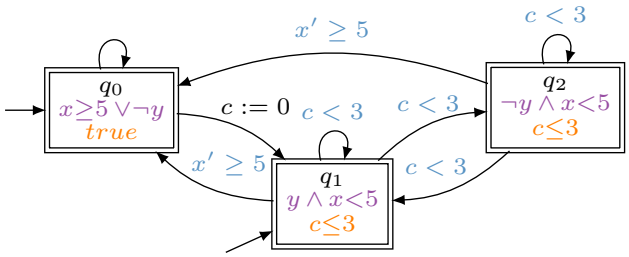


Fig. 1: PEA $\mathcal{A}_{r_0}$ equivalent to requirement r_0 (similarly for r_1 with all time bounds showing five instead of three). Location invariants are coloured purple, clock invariants orange, and guards blue. Guards evaluating to *true* are omitted.

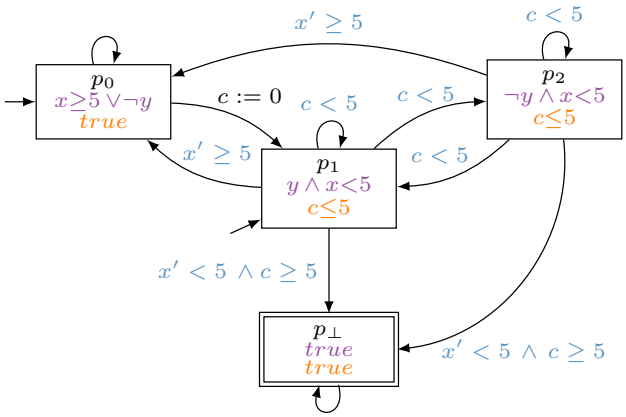


Fig. 2: Complement $\mathcal{A}^c_{r_1}$ of the PEA equivalent to r_1. Locations with a single border are considered to be non-accepting.

checks subsumption of any requirement in a set of requirements against the remaining set.

The automatic detection of subsumption that we present in this paper is based on an automaton representation of the requirements. Each requirement in HANFORPL has an equivalent, so-called Phase Event Automaton (PEA). A PEA is a timed automaton over system variables (here a numeric x and a Boolean y) and clocks to measure time (c in Fig. 1). A PEA equivalent to a requirement accepts exactly the executions, i.e., sequences of valuations of system variables and their duration, that are permitted by the requirement. An example PEA of r_0 is shown in Fig. 1.

A run of a PEA is a sequence of configurations. Each configuration is a tuple of a location in the PEA, a valuation of system variables, a valuation of clocks to track time, and the duration spent in the current configuration. For example, a concrete (initial) configuration $(q_0, \beta_0, \gamma_0, t_0)$ of $\mathcal{A}_{r_0}$ starts in the initial location q_0 (signified by the inbound arrow), has a valuation of system variables β_0 that fulfils the location invariant $x \geq 5 \vee \neg y$, e.g., $\beta_0 = \{x \mapsto 1, y \mapsto \mathit{false}\}$, has a valuation of clocks γ_0 that fulfils the clock invariant *true*, e.g., $\gamma_0 = \{c \mapsto 0\}$, and a duration t_0 that does not extend any clock to violate a clock invariant, e.g., $t_0 = 4.0$. As the clock invariant is *true*, t_0 does not have an upper bound in q_0.

To choose a successor configuration, we can either stay in q_0 by the self-loop, by choosing a similar valuation for variables and clocks, or we transition to location q_1, for example by choosing $\beta_1 = \{x \mapsto 1, y \mapsto \mathit{true}\}$. In this example, we do the latter. Although time advancing by 4, the clock valuation is $\gamma_1 = \{c \mapsto 0\}$ as the edge between q_0 and q_1 resets the clock ($c := 0$). Location q_1 has the clock invariant $c \leq 3$, the duration t_1 has to be smaller than or equal to 3 as any higher number causes the clock to exceed the clock invariant. We choose $t_1 = 3$.

Any further extension of the current run has to set the variable x such that $x \geq 5$. This is because all edges leaving q_1 either are guarded by $c < 3$ guaranteeing that in any successor configuration there remains sufficient (i.e., non-zero) time to fulfil the requirement, which is already exceeded (as $\gamma(c)_1 + t_1 = 3$), or requires $x \geq 5$ as the only remaining edge is guarded by $x' \geq 5$ (here x' refers to the value of x in the successor configuration).

To check redundancy of a requirement within a set of requirements, we have to prove that the requirements set with the requirement in question being removed is still a model for the requirement in question. Or, in other words, to show that it is not possible to fulfil the negation of the requirement in the remainder of the requirements set. To analyse a set of requirements for redundancy of any requirement, we use the classical approach to automata-based model checking [12]. In this approach, the model is intersected with the negation of the property that is to be checked. If the intersection is empty, there exists no execution fulfilling the model and the property. If not, any word in the intersection is a counter example proving violation of the given property.

In contrast to the classical approach, we cannot use the negation of a requirement, but have to calculate the complement of the PEA as shown in Section III. An example of a complement PEA of requirement r_1 is shown in Fig. 2. While, in a classical PEA, all locations can be seen as accepting locations, an explicit acceptance condition has to be introduced when complementing PEAs: a run is only accepted if it ends in a location that is accepting (locations with a double border) and is not accepted otherwise. Intuitively, the introduction of an acceptance condition allows the complement PEA to wait until something bad (i.e. a violation of the requirement equivalent to the PEA) happens, accepting any continuation of this bad prefix. For example, a run reaching location p_1 or p_2 (Fig. 2) that will exceed the clock invariant in the next configuration is not required to fulfil $x \geq 5$. It is able to transition to $p_\perp$, violating the requirement, and is therefore accepted in the complement automaton. As the accepting location only has a self loop without guard, and both the location and clock invariant are *true*, any continuation from here on will also be accepted.

To demonstrate the redundancy check, we now observe the intersection of the automata $\mathcal{A}_{r_0}$ and $\mathcal{A}^c_{r_1}$ (Fig. 1 and Fig. 2). We continue using the run $((q_0, p_0), \beta_0, \gamma_0, t_0), ((q_1, p_1), \beta_1, \gamma_1, t_1)$ from the initial example. In the intersection, each location is

a tuple of the respective automata. The valuations of clocks and automata are checked against the location invariants, clock invariants and guards of the respective locations and transitions of each automaton. In the intersection, location (q_1, p_1) is reachable. If requirement r_1 is not redundant with r_0, a run has to exist, that reaches $p_\perp$, as it would be a violation of r_1 that is not already prohibited by r_0. All transitions to $p_\perp$ are guarded by $x' < 5 \wedge c \geq 5$. Contrary, a continuation of the above run in $\mathcal{A}_{r_0}$ is only possible if $x \geq 5$ as long as $c \leq 3$, i.e., there does not exist any extension of the run, that enters $p_\perp$. As $p_\perp$ is not reachable, there is no run in the intersection, and therefore r_1 does not have any influence on the described behaviour of the requirements set $\{r_1, r_0\}$.

The above search for an accepted word in the intersection can be reduced to a program analysis task where in a program, encoding the intersected automata, reachability of a statement is shown (Section IV). Scalability of this approach is demonstrated in the benchmarks in Section V.

II. Preliminaries

As shown in Fig. 1 and 2, locations and transitions of PEAs are labelled with formulas. Before formally introducing PEAs, we define some notational shortcuts that are useful in this context:

By $\mathcal{L}(V)$, we denote the language of Boolean expressions with free variables in V.

A *clock constraint* δ_c is a formula of the form $c < t$ or $c \leq t$. By $\mathcal{L}(C)$, we denote the language of Boolean expressions with clock constraints on clocks in C. A clock invariant I is the conjunction of clock constraints, i.e., $I = \bigwedge \delta_c$. By $I_<(p)$ and $I_\leq(p)$, we denote the two functions that transform the clock invariant I of a location p into its strict and non-strict version, respectively. For example, for $I(p) = c_0 < t_0 \wedge c_1 \leq t_1$, we get $I_<(p) = c_0 < t_0 \wedge c_1 < t_1$ and $I_\leq(p) = c_0 \leq t_0 \wedge c_1 \leq t_1$. With a slight abuse of notation, we write $\delta_c \in I(p)$ to denote that clock constraint δ_c appears as a conjunct in the clock invariant of location p.

We write $F = \{s \mapsto B(s) \mid s \in S\}$ to denote a function that maps each element in a set S to the Boolean predicate $B(s)$. By $\neg F$, we then abbreviate the function mapping each element in S to the negation of $B(s)$, i.e., $\neg F = \{s \mapsto \neg B(s) \mid s \in S\}$

To denote the set of all outgoing transitions of location p, we write $E(p)$.

A **Phase Event Automaton** (PEA) is defined as a tuple $\mathcal{A} = (P, V, C, E, s, I, E_0)$ with the following components[1]:

- P being a finite set of locations,
- V being a finite set of typed variables,
- C being a finite set of clock variables,
- $E \subseteq P \times \mathcal{L}(V \cup C \cup V') \times 2^C \times P$ being a set of edges. An edge is a tuple (p, g, X, p'), where p and p' represent source and target location, guard g is a Boolean expression over primed variables in V and unprimed clock variables in C, and $X \subseteq C$ is a set of clocks to be reset,
- $s : P \to \mathcal{L}(V)$ is a function labelling each location with a Boolean expression over variables in V, called the location invariant,
- $I : P \to \mathcal{L}(C)$ is a function labelling each location with a Boolean expression over clock constraints, called the clock invariant,
- E_0 being a set of initial edges of the form (g, p') with guard g and target location p'.

We call a location $p_0 \in P$ an *initial* location if there is at least one initial edge $e \in E_0$ such that p_0 is the target location.

A **configuration** of a PEA is a tuple (p, β, γ, t) with location $p \in P$, variable valuation β for all variables $v \in V$, clock valuation γ for all clock variables $c \in C$, and a non-zero duration t.

A **run** of a PEA is a finite sequence of configurations $\sigma = (p_0, \beta_0, \gamma_0, t_0), ..., (p_n, \beta_n, \gamma_n, t_n)$. A run σ is **accepting** if there is an initial edge $(g, p_0) \in E_0$ such that

- the initial guard g is satisfied by the initial valuation β'_0 for primed variables and the initial clock valuation $\gamma_0(c) = 0$, i. e., $\beta'_0, \gamma_0 \models g$,

and for each configuration

- the valuation satisfies the location invariant, i.e., $\beta_i \models s(p_i)$,
- the clock valuation increased by the duration satisfies the location's clock invariant, i. e., $\gamma_i + t_i \models I(p_i)$,

and for every consecutive pair of configurations, there is an edge $(p_i, g, X, p_{i+1}) \in E$ such that

- the guard g is satisfied by the valuation β_i for unprimed variables, β'_{i+1} for primed variables, and the valuation $\gamma_i + t_i$ for clock variables, i. e., $\beta_i, \beta'_{i+1}, \gamma_i + t_i \models g$,
- the later clock valuation resets all clocks in X and increments all others by duration t_i, i. e., $\gamma_{i+1}(c) := 0$ if $c \in X$ and $\gamma_{i+1}(c) := \gamma_i(c) + t_i$ otherwise.

A sequence of configurations is **stutter-free** if it cannot be reduced by merging consecutive configurations. Two consecutive configurations $(p_i, \beta_i, \gamma_i, t_i)$ and $(p_{i+1}, \beta_{i+1}, \gamma_{i+1}, t_{i+1})$ can be merged to $(p_i, \beta_i, \gamma_i, t_i + t_{i+1})$ if their location and variable valuations coincide, i.e., if $p_i = p_{i+1}$ and $\beta_i = \beta_{i+1}$.

The **language** $\mathcal{L}(\mathcal{A})$ of a PEA $\mathcal{A}$ is defined as the set of all sequences of valuations and durations $\pi = (\beta_0, t_0), \ldots, (\beta_n, t_n)$ corresponding to an accepting run $(p_0, \beta_0, \gamma_0, t_0), ..., (p_n, \beta_n, \gamma_n, t_n)$ in $\mathcal{A}$.

We call a PEA **deterministic** if and only if it satisfies the following two conditions:

1) for each valuation β', there is at most one initial edge $(g, p) \in E_0$ such that $\beta' \models g$ and $\beta' \models s'(p)$;
2) for each location $p \in P$ and pair of valuations β' and γ, there is at most one edge $(p, g, X, p') \in E$ such that $\beta', \gamma \models g$ and $\beta' \models s'(p')$ and $\gamma[X := 0] \models I_<(p')$

The **parallel composition** [14] of two PEAs $\mathcal{A}_1 || \mathcal{A}_2$ is defined as the tuple $(P_1 \times P_2, V_1 \cup V_2, C_1 \dot\cup C_2, E, s_1 \wedge s_2, I_1 \wedge I_2, E_0)$ with $((p_1, p_2), g_1 \wedge g_2, X_1 \cup X_2, (p'_1, p'_2)) \in E$ and $(g_1 \wedge g_2, (p_1, p_2)) \in E_0$ for every pair of edges in E_1 and E_2, and $E_{0,1}$ and $E_{0,2}$, respectively. Similarly, we build the parallel composition for more than two PEAs.

[1]We slightly modify the definition of PEAs from [13] to use a set of initial edges instead of a set of initial locations, as we do not assume initial guards to be trivially *true*.

We define our notion of redundancy in the style of [15]: A requirement $r \in \mathcal{R}$ is **redundant** in a set of requirements $\mathcal{R}$, if

$$\bigwedge(\mathcal{R} \setminus r) \models r$$

i.e., the requirements set $\mathcal{R} \setminus r$ is a model for r without containing r itself. In other words, r does not impose any additional restriction on the models fulfilling than $\mathcal{R} \setminus r$ already does.

For the remainder of this paper, we assume formal requirements to be written in the formal requirements pattern language HANFORPL. For brevity, we refer the reader to [3], [10] for further detail. Also, we will refer to the requirements only by their equivalent PEA i.e., the PEA accepting the same valuations of system variables over the same durations as the requirement.

III. Redundancy Checking

In the following, a set of requirements is represented by a set of PEAs $\mathcal{R} = \{\mathcal{A}_1, ..., \mathcal{A}_m\}$. Thus, to restate the definition of *redundancy* in relation to automata: A requirement $\mathcal{A}_i$ is *redundant* in $\mathcal{R} \setminus \{\mathcal{A}_i\}$, if it can be omitted, i.e., the system behaviour expressed by the PEAs in the set $\mathcal{R}$ is equivalent to the system behaviour expressed by the PEAs in the set $\mathcal{R} \setminus \{\mathcal{A}_i\}$. This means that there exist PEAs $\mathcal{A}_j$ with $j \neq i$ in $\mathcal{R}$ such that

$$\bigcap_{\{\mathcal{A}_j \in \mathcal{R} | j \neq i\}} \mathcal{L}(\mathcal{A}_j) \subseteq \mathcal{L}(\mathcal{A}_i).$$

In order to check a set of requirements for redundancy, we restate *redundancy* as a classical automata theoretical model checking problem [12]:

$$\overline{\mathcal{L}(\mathcal{A}_i)} \cap \bigcap_{\{\mathcal{A}_j \in \mathcal{R} | j \neq i\}} \mathcal{L}(\mathcal{A}_j) = \emptyset.$$

If this intersection is empty, $\mathcal{A}_i$ is *redundant* since all executions that would be prohibited by $\mathcal{A}_i$ are already prohibited by the rest of the set $\mathcal{R}$ (cf. example in Section I-A). If this intersection is non-empty, there are executions exclusively prohibited by $\mathcal{A}_i$. In that case, $\mathcal{A}_i$ is *non-redundant*.

In order to perform the above emptiness check, a complement procedure for PEAs is required. The classical approach of negation of the underlying logical formula and transformation of the negated formula into an automaton is not feasible here, as the underlying fragment of Duration Calculus [16] is not closed under negation [14]. In the following, we present procedures to totalize and then complement PEAs. Both procedures preserve determinism of the input automaton.

A. Canonical Approach to PEA Totalization

The PEAs used to represent requirements reject any sequences of valuations and durations that would violate the requirement, so that any run of the automaton is an accepting run. For the complementation of a PEA, we need to capture both the accepted and unaccepted behaviour. We hence introduce the notion of a total PEA.

Definition 1 (Total PEA)**.** *A PEA $\mathcal{A} = (P, V, C, E, s, I, E_0)$ is called* total *if and only if it satisfies the following two conditions:*

1) *for each valuation β', there is exactly one initial edge $(g, p) \in E_0$ such that $\beta' \models g$ and $\beta' \models s'(p)$;*
2) *for each location $p \in P$ and pair of valuations β' and γ, there is exactly one edge $(p, g, X, p') \in E$ such that $\beta', \gamma \models g$ and $\beta' \models s'(p')$ and $\gamma[X := 0] \models I_<(p')$.*

In other words, a PEA $\mathcal{A}$ is total if for every sequence of valuations and durations $(\beta_0, t_0), \ldots, (\beta_n, t_n)$ there exists a unique, minimal sequence of configurations $\sigma = (p_0, \beta_0, \gamma_0, t_0), ..., (p_n, \beta_n, \gamma_n, t_n)$ such that σ is a run of $\mathcal{A}$. Note that, by definition, a total PEA is also deterministic.

To build a total PEA, we introduce a sink location $p_\perp$ with the property that incoming transitions are only enabled, if the requirement is violated. That is, from each location $p \neq p_\perp$, there is a transition to the sink location of the form $(p, g_\perp, \emptyset, p_\perp)$. As $I(p_\perp) = \mathit{true}$, there is no need for any further clock reset, i.e., $X = \emptyset$. The guard $g_\perp$ for a transition from p to $p_\perp$ is defined as:

$$g_\perp(p) := \neg \bigvee_{(p,g,X,p') \in E} \left(g \wedge s'(p') \wedge \bigwedge_{\{\delta_c | \delta_c \in I_<(p') \wedge c \notin X\}} \delta_c \right).$$

This guard ensures that $p_\perp$ is only reachable when none of the original outgoing edges can be taken.

The set of all non-initial sink transitions is then given as $\bigcup_{p \in P} (p, g_\perp(p), \emptyset, p_\perp)$. Similarly, we define the guard $g_\perp^{in}$ for the initial transition to the sink location as

$$g_\perp^{in} := \neg \bigvee_{(g,p) \in E_0} (g \wedge s'(p)).$$

This guard ensures that the sink location can only be entered initially, if none of the original initial locations can be entered.

As an example, consider again PEA $\mathcal{A}_{r_1}$ from our running example. The respective total PEA is shown in Fig. 3, where the parts coloured in black correspond to the original PEA, and the extensions made for the totalization is coloured in red. It contains the sink location $p_\perp$ where both the location and clock invariant are set to *true*, such that any continuation of a run is accepted. The sink location $p_\perp$ cannot be entered initially since its initial guard $g_\perp^{in} = \neg(x' \geq 5 \vee \neg y' \vee (y' \wedge x' < 5))$ evaluates to *false*. Every location has a transition to the sink location labelled by the corresponding guard. For p_0, we have

$$\begin{aligned} g_\perp(p_0) = \\ \neg((\mathit{true} \wedge (x' \geq 5 \vee \neg y') \wedge \mathit{true}) \vee (y' \wedge x' < 5)) \\ = \mathit{false}. \end{aligned}$$

The guards $g_\perp(p_1)$ and $g_\perp(p_2)$ are computed analogously. In the graphical representation of a PEA, we usually omit transitions whose guard evaluates to *false*.

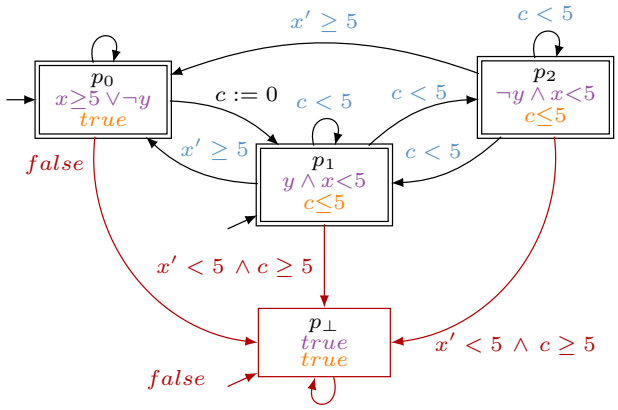


Fig. 3: Totalized version of PEA $\mathcal{A}_{r_1}$ representing r_1.

By definition, any location of a PEA is accepting. However, if we simply assume that the sink location is not accepting, then any run reaching the sink location is a non-accepting run. Introduction of a simple sink location is possible as requirements expressed by PEAs may only be safety requirements, i.e., requirements that prevent anything bad from happening (in contrast to liveness which is only fulfilled by a good thing eventually happening) [12]. Because of the safety limitation, any extension of a run reaching the sink location is still a non-accepting run, thus any further extension has to stay in the sink location. It is hence not possible to leave the sink location once entered; the only outgoing edge is a self-loop whose guard evaluates to *true*. This edge is required for the parallel composition, i.e., to allow a product automaton to take a step although there exist component automata remaining in the same location. Further, as the prefix is already violating the property, the automaton may now just accept any valuation, therefore, both the state and clock invariant of the sink location are $true$.

Under the given assumption that the sink location is not accepting, the language of the totalized PEA is equivalent to the language of the original one, while not rejecting any sequences of valuations and duration.

B. PEA Totalization

The totalization as described in the previous section is quite canonical. However, for locations with strict clock invariants, some runs are accepted erroneously. As an example, consider the requirement

r_2: *Once y holds, it holds for less than 5 time units.*

represented by the PEA in Fig. 4a. The PEA is called *strict*, since location p_1 contains the strict clock constraint $c < 5$ in its clock invariant. This strict clock constraint ensures that the duration of any interval in which y holds is strictly smaller than 5 time units. The strict clock invariant $c < 5$ enforces that location p_1 is left before $c = 5$ holds. The totalized PEA, following the procedure from Section III-A, is shown in Fig. 4b. Here, the sink location $p_\perp$ is never reachable. The guard of its sink transition, $c \geq 5$, is never satisfied as the strict clock invariant $c < 5$ enforces that location p_1 is left before $c = 5$ holds.

We prevent non-accepting runs of strict PEAs to get stuck by replacing the strict clock constraint (after performing the hitherto described totalization) by its non-strict version (Fig. 4c). The sink location is now reachable; any run where y holds for more than 5 time units is captured within the sink location. However, even when assuming that the sink location is non-accepting, the language of the modified PEA is a superset of the language of the original one. Any run in which the observable y holds for exactly 5 time units, e. g., the run $(p_1, \{y \mapsto true\}, \{c \mapsto 0\}, 5)$, is erroneously accepted by the total PEA with the modified clock constraint.

We prevent this by introducing the notion of a conditional acceptance criterion for locations with strict clock invariants and combine it with the previous modification (Fig. 4d). The respective location is only accepting as long as the strict clock invariant holds and then gets non-accepting once the clock equals the exact clock limit, allowing to reach the sink location afterwards. In Fig. 4d, the conditional acceptance criterion for location p_1 is indicated by splitting the clock invariant into the strict clock invariant $(c < 5)$ for which the location is considered to be accepting, and the equality $c = 5$ for which the location is considered to be non-accepting.

In a last step, we have to explicitly conjoin $c < 5$ to the guard of all outgoing transitions of p_1 that carry no constraint for the clock of the modified constraint (Fig. 4e). Before the modification, this constraint was implicitly given by the clock invariant, since the location had to be left before the limit of the clock constraint was reached.

With this extended totalization procedure and under the assumption that the sink location is non-accepting, the language of the totalized PEA $\mathcal{A}^t_{r_2}$ is equivalent to the language of the original PEA $\mathcal{A}_{r_2}$.

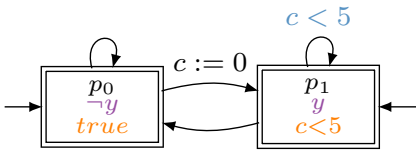
Since we need to distinguish accepting and non-accepting locations, possibly in dependence of the clocks, we introduce the notion of *conditional PEAs* (CPEAs).

Definition 2 (Conditional PEA)**.** *A* conditional PEA $\mathcal{A} = (P, V, C, E, s, I, E_0, F)$ *is a PEA* (P, V, C, E, s, I, E_0) *where* $F : P \to \mathcal{L}(C)$ *is a function labelling each location with a Boolean expression over clock constraints over* C*.*

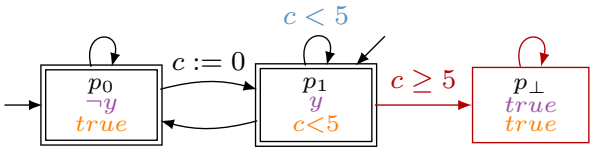
We call the clock constraint introduced by F the *acceptance condition*. The concept of totality for PEAs (Def. 1) can also be applied to CPEAs. We refine the definition of an accepting run for PEAs (cf. Section II) by adding the constraint that the final location in a run must be accepting at the end of the current configuration.

Definition 3 (Accepting CPEA run)**.** *Let* $\mathcal{A} = (P, V, C, E, s, I, E_0, F)$ *be a deterministic CPEA and let* $\sigma = (p_0, \beta_0, \gamma_0, t_0), ..., (p_n, \beta_n, \gamma_n, t_n)$ *be a stutter free run of* $\mathcal{A}$*. Run* σ *is accepting, if and only if it is an accepting run of the PEA* (P, V, C, E, s, I, E_0) *and if*

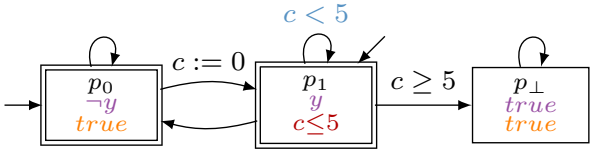
$$\gamma_n + t_n \models F(p_n).$$



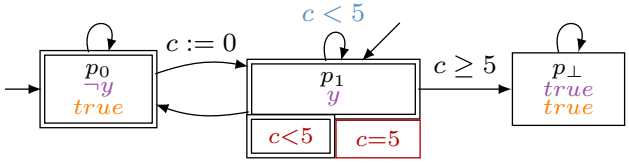
(a) Strict PEA $\mathcal{A}_{r_2}$ representing the requirement *"Once y holds, it holds for less than 5 time units"*.



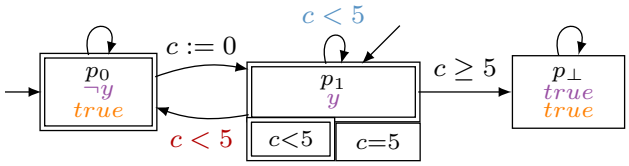
(b) Totalization of $\mathcal{A}_{r_2}$, yielding an unreachable sink location.



(c) Totalization of $\mathcal{A}_{r_2}$ with subsequently modified clock constraint, yielding incorrect language representation.



(d) Totalization of $\mathcal{A}_{r_2}$ with modified clock constraint and conditional acceptance condition.



(e) Totalization of $\mathcal{A}_{r_2}$ with modified clock constraint, conditional acceptance condition, and modified outgoing transition.

Fig. 4: Totalization steps of a strict PEA.

Note, that with $F(p) = true$ for all $p \in P$, the language of the CPEA $\mathcal{A} = (P, V, C, E, s, I, E_0, F)$ is equivalent to the language of the PEA $\mathcal{A}' = (P, V, C, E, s, I, E_0)$.

In the following, we show how CPEAs can be used to construct totalized PEAs such that the language of the totalized PEA and the language of its original version are equivalent. Our totalization procedure involves the following steps to transform a deterministic PEA into a totalized CPEA:

1) Introduction of a sink location.
2) Computation of initial sink transition $g_\perp^{in}$.
3) Computation of sink transition $g_\perp(p)$ for $p \in P$.
4) Replacement of strict clock constraints by their non-strict version.
5) Modification of outgoing transitions for locations with modified clock constraint.
6) Computation of acceptance conditions.

The resulting totalized CPEA is formally defined as:

Definition 4 (Totalized PEA)**.** *Given a deterministic PEA $\mathcal{A} = (P, V, C, E, s, I, E_0)$, its totalized CPEA is given as $\mathcal{A}^t = (P^t, V, C, E^t, s^t, I^t, E_0^t, F)$ where*

- $P^t := P \cup \{p_\perp\}$,
- $E^t := (\bigcup_{p \in P} E^t(p)) \cup E_\perp$,
- $s^t := s \dot\cup \{p_\perp \mapsto true\}$,
- $I^t := \{p \mapsto I_\leq(p) \mid p \in P\} \dot\cup \{p_\perp \mapsto true\}$,
- $E_0^t := E_0 \cup \{(g_\perp^{in}, p_\perp)\}$, *and*
- $F := \{p \mapsto \bigwedge_{(c_i < t_i) \in I(p)} c_i < t_i \mid p \in P\} \dot\cup \{p_\perp \mapsto false\}$

where $I_\leq(p)$ is the non-strict version of the clock invariant $I(p)$ and where the transitions of the CPEA are defined as

$$E^t(p) := \begin{cases} E(p) & \text{if } I(p) = I_\leq(p), \\ \{(p, g^t, X, p') \mid (p, g, X, p') \in E \\ \quad \wedge\, g^t = g \wedge \bigwedge_{(c_i < t_i) \in I(p)} c_i < t_i\} & \text{otherwise,} \end{cases}$$

and guards on the sink transitions are defined as

$$g_\perp(p) := \neg \bigvee_{(p,g,X,p') \in E} \left(g \wedge s'(p') \wedge \bigwedge_{\{\delta_c \mid \delta_c \in I_<(p') \wedge c \notin X\}} \delta_c \right).$$

and the transitions to the sink location are defined as

$$E_\perp := \bigcup_{p \in P} (p, g_\perp(p), \emptyset, p_\perp) \cup \{(p_\perp, true, \emptyset, p_\perp)\}$$

and with initial guards of the sink location defined as

$$g_\perp^{in} := \neg \bigvee_{(g,p) \in E_0} (g \wedge s'(p)).$$

For a totalized CPEA, we fix the acceptance condition F such that it maps the original locations in P with strict clock constraints to $true$, those with non-strict constraints to the conjunction of all modified clock constraints, and the sink location $p_\perp$ to $false$.

If a location's clock invariant contains a combination of strict and non-strict clock constraints, the clock invariant of the accepting part will be the original clock invariant, while the clock invariant of the non-accepting part is the disjunction $\bigvee c_i = t_i$ of all non-strict clock constraints $c_i < t_i$ in the original clock invariant. For example, for location p with clock invariant $I(p) = c_1 < 3 \wedge c_2 \leq 4 \wedge c_3 < 5$, the accepting part considers $I_<(p)$ while the non-accepting part considers $c_1 = 3 \vee c_3 = 5$.

Theorem 1. *Given a deterministic PEA $\mathcal{A} = (P, V, C, E, s, I, E_0)$, the totalized PEA $\mathcal{A}^t$ (given in Def. 4) is total, and language preserving (i.e., $\mathcal{L}(\mathcal{A}^t) = \mathcal{L}(\mathcal{A})$).*

Proof. Let $\mathcal{A} = (P, V, C, E, s, I, E_0, F)$ be a deterministic PEA and $\mathcal{A}^t = (P^t, V, C, E^t, s^t, I^t, E_0^t, F)$ its totalized PEA.

We show totality of $\mathcal{A}^t$ by showing that there is exactly one initial transition enabled for every valuation β' and that in each location $p \in P^t$ there is exactly one transition enabled for every pair of valuations β' and γ.

As $\mathcal{A}$ is deterministic, for each location $p \in P$ (and initially), there is at most one transition enabled for every pair of valuations β' and γ. By definition of $g_\perp^{in}$, the initial sink transition is enabled exactly if none of the original initial transitions is enabled, i.e., $g_\perp^{in} \Leftrightarrow \neg \bigvee_{(g,p)\in E_0} (g \wedge s'(p))$.

For outgoing transitions of location p, we make a case distinction on the type of p. For $p = p_\perp$, the only outgoing transition $(p_\perp, \mathit{true}, \emptyset, p_\perp)$ is enabled for all β', γ.

For $p \in P$: By definition of $g_\perp(p)$, the guard of the sink transition $(p, g_\perp(p), \emptyset, p_\perp)$ is satisfied exactly if none of the original outgoing transitions of p is enabled, i.e.,

$$g_\perp(p) \Leftrightarrow$$
$$\neg \bigvee_{(p,g,X,p')\in E} \left(g \wedge s'(p') \wedge \bigwedge_{\{\delta_c | \delta_c \in I_<(p') \wedge c \notin X\}} \delta_c \right).$$

For non-strict locations, this coincides with the sink transition being enabled. However, for a strict location, this is not the case, as the strict clock invariant enforces that the location is left before the sink transition can be taken. By changing the strict clock invariant to its non-strict version, the sink transition can be taken. Implicit strict constraints are explicitly conjoined to the guards of all outgoing transitions that do not yet imply these strict constraints. Thus, after the modifications, the sink transition is enabled, exactly if none of the original outgoing transitions of p is enabled. There is hence exactly one (initial) transition enabled for all β' and γ. Therefore, $\mathcal{A}^t$ is total.

We now show that the totalization is language preserving: Let $\sigma = (p_0, \beta_0, \gamma_0, t_0), ..., (p_n, \beta_n, \gamma_n, t_n)$ be a stutter-free accepting run in $\mathcal{A}$. Then, $\gamma_i + t_i \models I(p_i)$ for all $i \in \{0, \ldots, n\}$, in particular, $\gamma_n + t_n \models I(p_n)$. Since $\mathcal{A}^t$ is total, σ must be a run in $\mathcal{A}^t$. To show that σ is accepting in $\mathcal{A}^t$, we make a case distinction on the strictness of location p_n. If p_n is a non-strict location, then $F(p) = \mathit{true}$, hence $\gamma_n + t_n \models F(p_n)$. Thus σ is accepting in $\mathcal{A}^t$. Otherwise, if p_n is a strict location, $F(p_n) = \bigwedge_{(c_i<t_i)\in I(p)} c_i < t_i$. By definition of the acceptance condition, $I(p) \rightarrow F(p)$ for all $p \neq p_\perp$. Thus, $\gamma_n + t_n \models F(p_n)$, and σ is accepting in $\mathcal{A}^t$. Therefore, $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}^t)$. For the second inclusion, let $\sigma = (p_0, \beta_0, \gamma_0, t_0), ..., (p_n, \beta_n, \gamma_n, t_n)$ be a stutter-free accepting run in $\mathcal{A}^t$. Then σ is a stutter-free accepting run in $\mathcal{A}$, and hence $\mathcal{L}(\mathcal{A}^t) \subseteq \mathcal{L}(\mathcal{A})$. Therefore, $\mathcal{L}(\mathcal{A}^t) = \mathcal{L}(\mathcal{A})$. □

Note, that the CPEA resulting from the described totalization procedure is still deterministic (as any total PEA is, by definition, deterministic).

C. PEA Complement

A total PEA as constructed by the procedure given in Section III-A contains both accepted and unaccepted behaviour. To complement a PEA, we hence only need to switch accepting and non-accepting locations. Fig. 5 shows the complement $\mathcal{A}^c_{r_2}$ of the PEA equivalent to r_2.

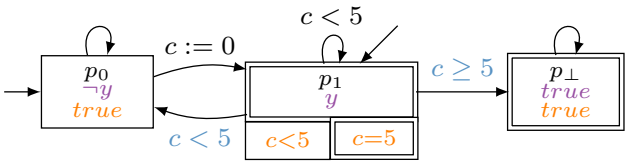


Fig. 5: Complement $\mathcal{A}^c_{r_2}$ of the PEA equivalent to r_2.

Theorem 2 (CPEA complement)**.** *Given a deterministic, totalized PEA* $\mathcal{A} = (P, V, C, E, s, I, E_0, F)$*, its* complement *automaton*

$$\mathcal{A}^c = (P, V, C, E, s, I, E_0, \neg F)$$

represents the complement language of $\mathcal{A}$*, i.e.,* $\mathcal{L}(\mathcal{A}^c) = \overline{\mathcal{L}(\mathcal{A})}$.

Proof. Let $\mathcal{A}^c = (P, V, C, E, s, I, E_0, \neg F)$ be the complement automaton of the totalized PEA $\mathcal{A} = (P, V, C, E, s, I, E_0, F)$.

By definition, the totalized PEA $\mathcal{A}$ does not reject any sequence of valuations and durations, such that all these sequences correspond to a run in $\mathcal{A}$. Since the totalization procedure preserves determinism, each sequence of valuations and durations corresponds to a unique stutter-free run.

A stutter-free run $\sigma = (p_0, \beta_0, \gamma_0, t_0), ..., (p_n, \beta_n, \gamma_n, t_n)$ is accepting in $\mathcal{A}$ if $\gamma_n + t_n \models F(p_n)$, and non-accepting if $\gamma_n + t_n \not\models F(p_n)$. Then, σ is accepting in the complement automaton $\mathcal{A}^c$ if $\gamma_n + t_n \models \neg F(p_n)$, i.e., if $\gamma_n + t_n \not\models F(p_n)$, and is non–accepting if $\gamma_n + t_n \not\models \neg F(p_n)$, i.e., $\gamma_n + t_n \models F(p_n)$.

Thus, exactly the runs not-accepting in $\mathcal{A}$ are accepting in $\mathcal{A}^c$ and vice versa. Therefore, $\mathcal{L}(\mathcal{A}^c) = \overline{\mathcal{L}(\mathcal{A})}$.

□

IV. Implementation

In this section, we give an overview of the implementation of our redundancy check.

Our approach is implemented in the program analysis framework Ultimate [17], developed at the University of Freiburg as part of the requirements analysis tool Ultimate ReqCheck. The implementation builds on an existing tool chain used for the analysis of *consistency*, *vacuity* [18] and *rt-consistency* [19]. The tool chain comprises several modules: A parser for the pattern language HanforPL emitting PEAs. A transformer (*Pea2Boogie*) translating the PEAs into a program in the verification language Boogie [20]. The program is equivalent to the parallel product (i.e. intersection of languages) of the PEAs with added annotations relating to each analysis [13]. Reachability analysis of these annotations is performed by Ultimate Automizer [21]. The redundancy check is integrated into this requirement analysis tool chain as a fourth analysis option.

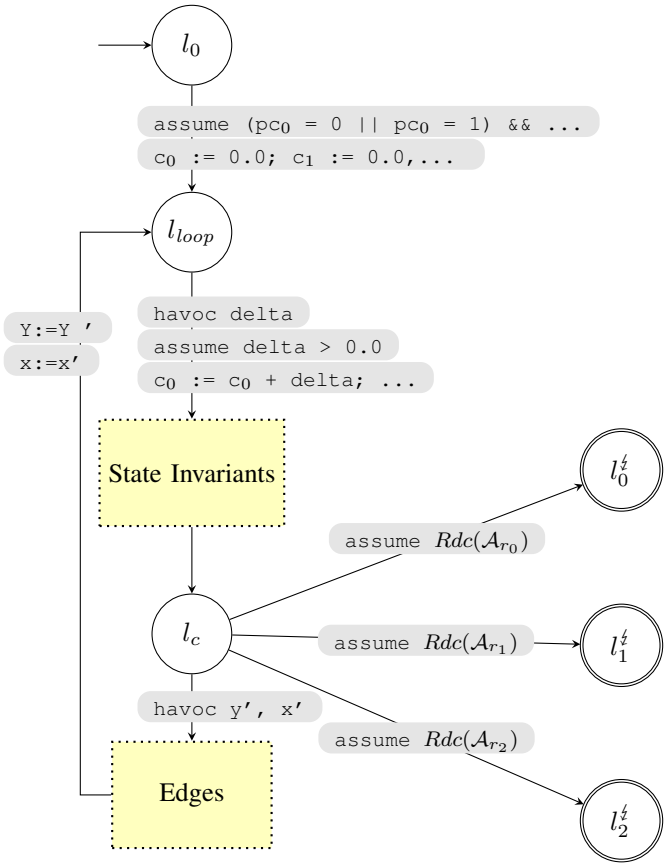


Fig. 6: Overall structure of the program $\mathcal{P}(\mathcal{A})$ encoding the parallel composition $\mathcal{A} = \mathcal{A}^t_{r_0}||\mathcal{A}^t_{r_1}||\mathcal{A}^t_{r_2}$

A. Intersection as a Boogie program

For a detailed explanation on the encoding, we refer the reader to [10], [13]. A schematic representation of the boogie encoding is shown in Fig. 6: States refer to relevant locations in the program, edges are labelled with the statement sequences between the locations and yellow boxes are sub programs abstracted for understandability. Error locations are marked by a double border and can be ignored for now.

At the beginning (l_0 to l_{loop}), the encoding initialises all variables: The clocks of all PEAs are set to zero, as no time has passed. The integer variables representing the current location of each PEA (e.g. $\texttt{pc}_{r_1}$) are set to an initial location. Note, that at this stage, all state variables of the PEAs (e.g. x) are non-deterministically assigned due to the boogie semantics, i.e. they have any possible value at once.

The encoding continues inside a loop starting in l_{loop} with each iteration referring to one configuration accepted by the current location of each PEA. The loop begins with choosing a positive non-zero duration (delta) for the configuration non-deterministically. This duration is then used to advance the clock of each PEA by the same amount.

In the following (sub program *State Invariants*), the program checks for each current location of each PEA, if the state invariant and clock invariant are satisfied. Because of the nondeterministic assignment, any run of the program reaching program location l_c is an initial configuration for the intersection of PEAs.

As transitions in a PEA may refer to the assignment of state variables in the successor configuration, primed copies of the state variables (e.g. x') are non-deterministically assigned a suitable value.

In the following (sub program *Edges*), the program checks if transitions can be taken, i.e., if the source location is the current location and if the guard is fulfilled. According to the transitions, variables representing the current locations of the PEAs are changed accordingly.

The loop ends by assigning the future values for all state variables to the variables encoding the current state variables.

The loop then continues as for the initial configuration, with any execution of the program reaching location l_c for the n-th time, encoding a run of length n of the PEA intersection.

B. Encoding Redundancy

To explain the encoding of the PEA intersection, we ignored the double bordered locations. When input to a program analysis tool, the tool will try to find a run of the program, that ends in these so-called *error locations*. For example, the error locations in Fig. 6 are guarded by assumptions over some logical formulas. An error location can be reached exactly if there is a program execution that reaches l_c and fulfils the logical formula assumed at the edge.

The program $\mathcal{P}(\mathcal{R})$ encoded for the redundancy check of the requirements $r \in \mathcal{R}$ is the parallel product of each totalized automaton

$$\mathcal{P}(\mathcal{R}) = \mathcal{A}^t_{r_1}||...||\mathcal{A}^t_{r_n}.$$

As totalized CPEAs do not reject on any input, the encoding thus far does accept arbitrary runs. Recalling the definition of redundancy detection by model checking (Section III), a requirement is redundant if the intersection of its complement and all other requirements is empty. This is encoded in one annotation per requirement in a redundancy condition.

Definition 5 (Redundancy Condition)**.** *Let $\mathcal{P}(\mathcal{R}) = \mathcal{A}^t_1||...||\mathcal{A}^t_n$ be an encoding of totalized CPEAs with components $\mathcal{A} = (P, V, C, E, s, I, E_0, F)$ and let $\texttt{pc}_\texttt{i}$ be the program variable representing the automaton location. The redundancy condition for the k-th automaton is defined as*

$$Rdc(\mathcal{A}_k) := \left(\bigvee_{p_j \in P_k} \texttt{pc}_\texttt{j} = p_j \wedge \neg F(p_j) \right) \wedge \bigwedge_{P_i \in \{P_1,...,P_n\} \setminus \{P_k\}} \left(\bigvee_{p_j \in P_i} \texttt{pc}_\texttt{j} = p_j \wedge F(p_j) \right).$$

As Theorem 2 shows, the complement of a CPEA is only dependent on the acceptance condition. Therefore, in the encoding, any automaton can serve as both, its totalized and its complement. In the redundancy condition, the first disjunction requires the k-th automaton to be the complement automaton by

negation of the acceptance condition, while any other automaton accepts normally.

To encode a redundancy analysis for each requirement, we add an error location and edge with the according redundancy condition to the monitor location l_c.

It is worth noticing that our implementation requires each automaton to be included only once in the encoding as a totalized automaton. The semantics of each automaton can simply be altered by changing the accepting locations in the respective assertion (see Theorem 2). This allows us to encode all redundancy analyses for a whole requirements set without much overhead (apart from the unused sink locations).

Theorem 3. *In a set of requirements $\mathcal{R}$, a requirement $r \in \mathcal{R}$ is not redundant, if in the encoding $\mathcal{P}(\mathcal{R})$ location $l_r^{\lightning}$ is reachable.*

The proof follows directly from model checking using automata (Section III-A) and the complement construction (Theorem 2). Existence of an execution of $\mathcal{P}(\mathcal{R})$ in which the complement of r_i as well as the remaining requirements accept the run is a witness for the non-emptiness of the intersection.

To clarify our approach, we now give an example of the encoding and redundancy condition for the running example. Let $\{\mathcal{A}^t_{r_0}, \mathcal{A}^t_{r_1}, \mathcal{A}^t_{r_2}\}$ be the set of totalized CPEAs for the example requirements. In Fig. 6, the overall structure of the program $\mathcal{P}(\mathcal{A})$ encoding the parallel composition $\mathcal{A} = \mathcal{A}^t_{r_0} || \mathcal{A}^t_{r_1} || \mathcal{A}^t_{r_2}$ is shown. As stated previously, it is not necessary to explicitly construct the complement PEAs or to include them in the parallel composition, since we can express the intersection by just negating the acceptance condition of any $\mathcal{A}^t_i$. The `assume` statement on the transition to location $l_i^{\lightning}$ expresses the redundancy condition for requirement r_i. For that, we assume that each location $p_0, \ldots, p_n$ of a PEA $\mathcal{A}^t$ is represented by the integers $0, \ldots, n$ and the sink location $p_\perp$ is represented by $n+1$. We take a closer look at $Rdc(\mathcal{A}_{r_0})$ on the edge to error location $l_0^{\lightning}$.

$$
\begin{aligned}
Rdc(\mathcal{A}_{r_0}) = {} & (\mathtt{pc}_{\mathtt{r}_0} = 3) \wedge && (1)\\
& (\mathtt{pc}_{\mathtt{r}_1} = 0 \vee \mathtt{pc}_{\mathtt{r}_1} = 1 \vee \mathtt{pc}_{\mathtt{r}_1} = 2) \wedge && (2)\\
& (\mathtt{pc}_{\mathtt{r}_2} = 0 \vee \mathtt{pc}_{\mathtt{r}_2} = 1 \wedge c_2 < 5) && (3)
\end{aligned}
$$

Line (1) ensures that $\mathcal{A}^t_{r_0}$ is in location 3, i.e., the sink location $p_\perp$. In $\mathcal{A}^t_{r_0}$, $p_\perp$ is the only non-accepting location. For p_0, p_1, and p_2, $\neg F(p_i)$ evaluates to *false* and $\neg F(p_\perp) = \mathit{true}$, so the disjunction over the locations of $\mathcal{A}^t_{r_0}$ simplifies to $(\mathtt{pc}_{\mathtt{r}_0} = 3)$. Line (2) ensures that $\mathcal{A}^t_{r_1}$ is in an accepting location. Since $\mathcal{A}^t_{r_1}$ is *strict*, $F(p) = \mathit{true}$ holds for every location p in $P \setminus \{p_\perp\}$, thus $\mathcal{A}^t_{r_1}$ is accepting if $(\mathtt{pc}_{\mathtt{r}_1} = 0 \vee \mathtt{pc}_{\mathtt{r}_1} = 1 \vee \mathtt{pc}_{\mathtt{r}_1} = 2)$. We can express the same with $(\mathtt{pc}_{\mathtt{r}_1} \neq 3)$, since the sink is the only non-accepting location in $\mathcal{A}^t_{r_1}$. Likewise, line (3) ensures that $\mathcal{A}^t_{r_2}$ is in an accepting location. In $\mathcal{A}^t_{r_2}$ we have location p_0 with $F(p_0) = \mathit{true}$ and p_1 with $F(p_1) = c < 5$. Therefore, $\mathcal{A}^t_{r_2}$ accepts if $(\mathtt{pc}_{\mathtt{r}_2} = 0 \vee \mathtt{pc}_{\mathtt{r}_2} = 1 \wedge c < 5)$ holds. Similarly, the redundancy conditions for $\mathcal{A}^t_{r_1}$ and $\mathcal{A}^t_{r_2}$ are computed as

$$
\begin{aligned}
Rdc(\mathcal{A}_{r_1}) = {} & (\mathtt{pc}_{\mathtt{r}_1} = 3) \wedge (\mathtt{pc}_{\mathtt{r}_0} \neq 3) \wedge \\
& (\mathtt{pc}_{\mathtt{r}_2} = 0 \vee \mathtt{pc}_{\mathtt{r}_2} = 1 \wedge c_2 < 5) \text{ and} \\
Rdc(\mathcal{A}_{r_2}) = {} & (\mathtt{pc}_{\mathtt{r}_2} = 1 \wedge c_2 \geq 5 \vee \mathtt{pc}_{\mathtt{r}_2} = 2) \wedge \\
& (\mathtt{pc}_{\mathtt{r}_0} \neq 3) \wedge (\mathtt{pc}_{\mathtt{r}_1} \neq 3).
\end{aligned}
$$

The error locations $l_i^{\lightning}$ are reachable if the `assume` statements are satisfied. Error location $l_0^{\lightning}$ is reachable with the run corresponding to the sequence of valuations and durations

$$
\begin{aligned}
\pi_0 = {} & (\{y \mapsto \mathit{false}, x \mapsto 3\}, 3), \\
& (\{y \mapsto \mathit{true}, x \mapsto 3\}, 4), \\
& (\{y \mapsto \mathit{true}, x \mapsto 6\}, 10).
\end{aligned}
$$

It represents system behaviour that is prohibited by r_0, since $x = 3$ holds for 4 time units after y changes its value to true, whereas r_0 enforces that $x \geq 5$ after at most 3 time units. The sequence π_0 is a word in $\mathcal{L}(\mathcal{A}_{r_1}) \cap \mathcal{L}(\mathcal{A}_{r_2})$, but not in $\mathcal{L}(\mathcal{A}_{r_0})$.

Similarly, error location $l_2^{\lightning}$ is reachable with the run corresponding to the sequence

$$
\begin{aligned}
\pi_2 = {} & (\{y \mapsto \mathit{false}, x \mapsto 3\}, 3), \\
& (\{y \mapsto \mathit{true}, x \mapsto 6\}, 6).
\end{aligned}
$$

This sequence represents prohibited system behaviour with respect to r_2, but permissible system behaviour with respect to r_0 and r_1. This means that requirements r_0 and r_2 are non-redundant. Location $l_1^{\lightning}$ is not reachable for any run, since there is no sequence of valuations that is a word in $\mathcal{L}(\mathcal{A}_{r_0}) \cap \mathcal{L}(\mathcal{A}_{r_2})$ but not in $\mathcal{L}(\mathcal{A}_{r_1})$. This means $\mathcal{R} \setminus \{r_1\}$ prohibits the same unwanted behaviour that r_1 prohibits. Therefore, r_1 is redundant and can be omitted from the set of requirements $\mathcal{R}$ without changing the specified system behaviour.

TABLE I: Redundancy analysis results of the replication package. Columns show an identifier (ID), requirements count (R), real time requirements count (RT), number of variables (V), non-redundant (NO), redundant (Yes), time out (TO) and total analysis time (T).

	Requirements			Redundancy			
ID	R	RT	V	No	Yes	TO	T (min)
dev-01	26	21	27	26	0	0	0.7
dev-02	50	47	53	49	1	0	5.8
dev-03	52	11	34	51	0	1	15.9
dev-04	58	53	53	57	1	0	7.2
dev-05	68	64	89	64	2	2	39.7
abz	83	52	52	78	5	0	23.3
dev-06	100	95	101	99	0	1	21.6
dev-07	107	80	172	107	0	0	3.5
dev-08	263	234	239	235	7	21	375.5
dev-09	407	358	326	396	4	7	464.1
dev-10	699	543	1003	589	7	8	819.2

V. Benchmarks

We analysed several sets of requirements for redundancies to evaluate our implementation of the redundancy check.

TABLE II: Results of the analysis of industrial requirements sets. Columns show an identifier (ID), requirements count (R), real time requirements count (RT), non-redundant (NO), redundant (Yes), time out (TO) and total analysis time (T). Some Bechmarks reached the global timeout (24 h), makred as TO followed by the number of remaining checks. Reruns with a lower per-check time limit are marked with *.

Requirements				Redundancy			
ID	R	RT	V	No	Yes	TO	T (min)
dev-a	45	37	68	44	1	0	3.9
dev-b	61	53	67	59	2	0	6.6
dev-c	62	30	76	62	0	0	2.8
dev-d	105	103	152	105	0	0	28.4
dev-e	115	112	156	115	0	0	35.6
dev-f	146	10	221	112	34	0	32.9
dev-g	185	151	180	256	4	2	45.2
dev-h	254	152	211	238	11	5	256.8
dev-i	368	328	383	78	37	68	TO (185)
dev-j	703	620	985	557	4	59	TO (63)
dev-k	763	394	561	437	55	29	TO (242)
dev-i*	368	328	383	141	35	192	1318.7
dev-j*	703	620	985	609	5	89	1034.7
dev-k*	763	394	561	612	34	117	1296.0

Requirements stem from industry co-operations, mostly in the the automotive domain. Formalisations were obtained from the industrial requirements as part of a formal analysis effort leading to, or following, the process described in [22]. The benchmarks were selected to cover requirements sets from different projects, as well as the whole range of sizes of requirements sets that were available to us. More precisely, requirements sets *dev-01* to *dev-10* (Table I) were from previous, publicly available benchmarks using requirements of BOSCH [13]. The remaining set (*abz* in Table I) is a previously formalised [10] set of pseudo realistic automotive requirements published by Houdek et al. [23] [2]. We also analysed more recent sets of industrial requirements (Table II), also from the automotive domain (not publicly available). We included these requirements sets as they more accurately reflect the current state of requirements pattern in HANFORPL and the experience using the available patterns and scopes.

Benchmarks were executed using Linux-5.15 with Java OpenJDK 11.0.18 64bit on an AMD Ryzen 5 5600 6-Core CPU with 3.5 GHz and 48 GB RAM. Benchmarks were performed using the benchmarking tool benchexec 3.21. ULTIMATE REQCHECK was run in version 0.2.3-4f54f8f5. Each analysis was assigned 30 GB of RAM, four cores and a 15 min timeout per redundancy check as well as one day (24 h) overall timeout.

Tables I and II show the benchmark results. Each table has an identifier for the requirements set (ID), and the number of requirements (R). Additionally, the subset of real-time requirements (RT) and the number of variables used in the requirements (V) is given, as they might impact the analysis time. Result columns show the number of redundancy analysis runs separated into redundancies found (Yes), proofs that a requirement is not redundant (No), and timeouts of the model checker (TO), as well as the total analysis time (T).

For our benchmarks, we assumed the analysis to be embedded in some process governing the formalisation and subsequent analysis of the requirements as described by Dietsch et al. [22]. As this, the analysis should be able to run within a nightly analysis cycle, and provide at least partial results to be acted upon. Based on these assumptions, our redundancy analysis performed well. All requirements sets could be analysed successfully or at least with partial results, leading to the detection of redundancies (and a high number of non-redundancy proofs). While no analysis exceeded the memory limitation, a number of timeouts could be observed.

Especially the analysis of set *dev-i* (Table II) was stopped after analysing approx. half of the requirements set due to the 24 h timeout. A closer inspection of the requirements set showed, that it is close to an adversarial example with global clocks forcing the analysis to deal with uncommonly long runs. Additionally, the majority of requirements is using complex patterns and scopes, resulting in complex behaviour for each individual requirement. Nonetheless, even in this set, a number of redundancies was found.

The ability to produce partial results even for a complex input is a strength of our approach (as already shown in [13]). Embedded in an analyse and fix cycle, the partial results enable improvements in the requirements. In this context, timeouts may vanish in later iterations due to changes in other requirements, or higher analysis times may be allotted to further increase requirements quality. To demonstrate the viability of this approach, we re-run benchmarks *dev-i* to *dev-k* with a per-check timeout of five minutes. This timeout seems to be a practical default setting in the beginning of a project (see sets marked with * in Table II).

Note, that we assume that all requirements are already given as formal requirements in HANFORPL. Therefore, any redundancy found is a real redundancy in the requirements and any requirement proven to be non-redundant is necessary for system behaviour (see Theorem 3). For a discussion of the relation of defects found in a formal requirements analysis and natural language requirements as a basis for this analysis, we refer to the process discussion in Dietsch et al. [22].

VI. RELATED WORK

Redundancy in requirements is listed as a threat to the modifiability by IEEE-830 [1] as well as Davis et al. [2]. Redundancies are intentionally not listed as defects, but emphasize the necessity for careful management of requirements that are stated multiple times.

The notion of redundancy can also be found in the model checking community. Kupferman [24] argues, that a successful run of a model checker, i.e., the model checker proved that a model does not violate the specification, can give false security. The successful verification of the model may be due to an ineffective specification rather than due to the model adhering to the specification. More precisely, by a specification that

[2] A replication package for benchmarks in Table I can be found at doi.org/10.5281/zenodo.10999174

itself is *vacuous* [25] or may look thorough but is really just bloated by repetition.

In the requirements community, there is some work on the detection of redundancies in natural language requirements: Jürgens et al. [26] applied duplication detection methods taken from source code analysis. Their experimentation on real world documents lead them to conclude that one should assume around ten percent of duplication if no precautions to eliminate redundancy were taken. This number is much higher than we could replicate in our benchmarks, but this may be due to requirements in our analysis being taken from a safety domain, as well as obvious redundancies already being eliminated during the formalisation process [22]. In contrast to our approach, the approach of Jürgens et al. is purely syntactical, i.e., unable to detect requirements that are less restrictive than a combination of other requirements.

There exists a number of tools for the analysis of formal requirements, e.g. [6]–[9]. None of these have the capability to detect arbitrary redundancies. While they are able to detect different properties, such as consistency, reliability and more, none of these tools have the ability to detect redundancies in the requirements set.

Dokhanchi et al. [27] preset checks to assist the elicitation of specifications in Metric Interval Temporal Logic (MITL), including the detection of redundancy but only within one formula, i.e., not detecting redundancy within a whole specification.

The work of Post et al. [18] handles vacuity in real time requirements, and is the basis for the current implementation of our vacuity analysis [13]. The detection of vacuity in addition to redundancy is useful although a vacuous requirement will also always be detected as redundant. As a vacuity is almost always a defect, redundancies caused by vacuity can directly be reported as defects, whereas the root cause of a redundancy has to be analysed carefully.

An approach to model checking PEAs was presented by Meyer et al. [28]. They use so-called Test Automata that share the concept of a sink location with CPAs. However, Test Automata are only defined for non-strict clock constraints. Also, their model checking approach is based on the direct translation of special Duration Calculus formulas into Test Automata.

VII. Conclusion

In this work, we presented a scalable approach to the redundancy analysis of real time requirements. This analysis is based on an automata theoretical approach that is enabled by the totalization and complement of an extension of the PEA timed automaton model that we introduce. We implemented the approach as a program analysis task solvable by reachability analysis. The benchmarks on industrial requirements sets show, that our approach is fit for practical use. The analysis successfully uncovered redundancies on large real world requirements sets from industry projects.

In future research we will apply the redundancy analysis to industrial projects to evaluate the reasons behind, as well as the impact of, redundant requirements in large requirements specifications.

Investigating redundancies reported by our analysis is still time consuming as the analysis currently only indicates the existence but not the cause. In the future, we will work on the automatic extraction of reasons for the redundancies found.

VIII. Acknowledgements

Funded by the Bundesministerium für Bildung und Forschung (BMBF, Federal Ministry of Education and Research, Germany) - 03VP11880 *SystemValid*.

References

[1] IEEE, *Recomm. Practice for Software Requirements Specifications*, 1993. 830:1998.

[2] A. M. Davis, S. Overmyer, K. Jordan, J. Caruso, F. Dandashi, A. Dinh, G. Kincaid, G. Ledeboer, P. Reynolds, P. Sitaram, A. Ta, and M. Theofanos, "Identifying and measuring quality in a software requirements specification," in *IEEE METRICS*, pp. 141–152, IEEE Computer Society, 1993.

[3] A. Post, I. Menzel, and A. Podelski, "Applying restricted english grammar on automotive requirements - does it work? A case study," in *REFSQ*, vol. 6606 of *Lecture Notes in Computer Science*, pp. 166–180, Springer, 2011.

[4] S. Ben-David, D. Fisman, and S. Ruah, "Temporal antecedent failure: Refining vacuity," in *CONCUR*, vol. 4703 of *Lecture Notes in Computer Science*, pp. 492–506, Springer, 2007.

[5] V. Langenfeld, A. Post, and A. Podelski, "Requirements defects over a project lifetime: An empirical analysis of defect data from a 5-year automotive project at bosch," in *REFSQ* (M. Daneva and O. Pastor, eds.), vol. 9619, pp. 145–160, Springer, 2016.

[6] T. Bienmüller, T. Teige, A. Eggers, and M. Stasch, "Modeling requirements for quantitative consistency analysis and automatic test case generation," in *FM&MDD*, 2016.

[7] A. W. Fifarek, L. G. Wagner, J. A. Hoffman, B. D. Rodes, M. A. Aiello, and J. A. Davis, "Spear v2.0: Formalized past LTL specification and analysis of requirements," in *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings* (C. W. Barrett, M. D. Davies, and T. Kahsai, eds.), vol. 10227 of *Lecture Notes in Computer Science*, pp. 420–426, 2017.

[8] A. Moitra, K. Siu, A. W. Crapo, M. Durling, M. Li, P. Manolios, M. Meiners, and C. McMillan, "Automating requirements analysis and test case generation," *Requir. Eng.*, vol. 24, no. 3, pp. 341–364, 2019.

[9] A. Katis, A. Mavridou, D. Giannakopoulou, T. Pressburger, and J. Schumann, "Capture, analyze, diagnose: Realizability checking of requirements in FRET," in *CAV (2)*, vol. 13372 of *Lecture Notes in Computer Science*, pp. 490–504, Springer, 2022.

[10] V. Langenfeld, *Formalisation and analysis of system requirements*. PhD thesis, University of Freiburg, Freiburg im Breisgau, Germany, 2023.

[11] E. Henkel, N. Hauff, L. Eber, V. Langenfeld, and A. Podelski, "An empirical study of the intuitive understanding of a formal pattern language," in *REFSQ*, vol. 13975 of *Lecture Notes in Computer Science*, pp. 21–38, Springer, 2023.

[12] C. Baier and J. Katoen, *Principles of model checking*. MIT Press, 2008.

[13] V. Langenfeld, D. Dietsch, B. Westphal, J. Hoenicke, and A. Post, "Scalable analysis of real-time requirements," in *RE*, pp. 234–244, IEEE, 2019.

[14] J. Hoenicke, *Combination of processes, data, and time*. PhD thesis, Carl von Ossietzky University of Oldenburg, 2006.

[15] H. Chockler and O. Strichman, "Before and after vacuity," *Formal Methods Syst. Des.*, vol. 34, no. 1, pp. 37–58, 2009.

[16] E. Olderog and H. Dierks, *Real-time systems - formal specification and automatic verification*. Cambridge University Press, 2008.

[17] D. Dietsch, *Automated verification of system requirements and software specifications*. PhD thesis, University of Freiburg, Freiburg im Breisgau, Germany, 2016.

[18] A. Post, J. Hoenicke, and A. Podelski, "Vacuous real-time requirements," in *RE*, pp. 153–162, IEEE Computer Society, 2011.

[19] A. Post, J. Hoenicke, and A. Podelski, "rt-inconsistency: A new property for real-time requirements," in *FASE*, vol. 6603 of *Lecture Notes in Computer Science*, pp. 34–49, Springer, 2011.

[20] K. R. M. Leino, "This is boogie 2," *manuscript KRML*, vol. 178, no. 131, p. 9, 2008.

[21] M. Heizmann, J. Hoenicke, and A. Podelski, "Software model checking for people who love automata," in *CAV*, vol. 8044 of *Lecture Notes in Computer Science*, pp. 36–52, Springer, 2013.

[22] D. Dietsch, V. Langenfeld, and B. Westphal, "Formal requirements in an informal world," in *FORMREQ*, pp. 14–20, IEEE, 2020.

[23] F. Houdek and A. Raschke, "Adaptive exterior light and speed control system," in *ABZ*, vol. 12071 of *Lecture Notes in Computer Science*, pp. 281–301, Springer, 2020.

[24] O. Kupferman, "Sanity checks in formal verification," in *CONCUR*, vol. 4137 of *Lecture Notes in Computer Science*, pp. 37–51, Springer, 2006.

[25] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient detection of vacuity in ACTL formulaas," in *CAV*, vol. 1254 of *Lecture Notes in Computer Science*, pp. 279–290, Springer, 1997.

[26] E. Jürgens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schätz, S. Wagner, C. Domann, and J. Streit, "Can clone detection support quality assessments of requirements specifications?," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010* (J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, eds.), pp. 79–88, ACM, 2010.

[27] A. Dokhanchi, B. Hoxha, and G. Fainekos, "Metric interval temporal logic specification elicitation and debugging," in *13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015*, pp. 70–79, IEEE, 2015.

[28] R. Meyer, J. Faber, J. Hoenicke, and A. Rybalchenko, "Model checking duration calculus: A practical approach," *Formal Aspects of Computing*, vol. 20, no. 4-5, pp. 481–505, 2008.