Electronic Communications of the EASST
Volume 076 (2019)

Automated Verification of Critical Systems 2018
(AVoCS 2018)

Analyzing Consistency of Formal Requirements

Jan Steffen Becker

26 pages

Guest Editors: David Pichardie, Mihaela Sighireanu
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Analyzing Consistency of Formal Requirements*

Jan Steffen Becker

becker@offis.de
OFFIS – Institute for Information Technology
Oldenburg, Germany

Abstract: In the development of safety-critical embedded systems, requirements-driven approaches are widely used. Expressing functional requirements in formal languages enables reasoning and formal testing. This paper proposes the Simplified Universal Pattern (SUP) as an easy to use formalism and compares it to SPS, another commonly used specification pattern system. Consistency is an important property of requirements that can be checked already in early design phases. However, formal definitions of consistency are rare in literature and tent to be either too weak or computationally too complex to be applicable to industrial systems. Therefore this work proposes a new formal consistency notion, called partial consistency, for the SUP that is a trade-off between exhaustiveness and complexity. Partial consistency identifies critical cases and verifies if these cause conflicts between requirements

Keywords: Formal Methods, Requirements Engineering, Consistency Analysis, Verification

1 Introduction

In designing safety critical embedded systems, requirements driven processes are widely used. These processes usual start with a textual description of the system requirements, that are further refined during the development process. For illustration purposes, imagine the design of a car's light system where a feature called tip-blinking is specified using the following requirements:

Req 1 If the pitman arm is moved down for less than 0.5s, left blinking shall be active for 3s.

Req 2 If the pitman arm is moved up for less than 0.5s, right blinking shall be active for 3s.

Furthermore we have normal blinking:

Req 3 If the pitman arm is moved down for at least 0.5s, left blinking shall be active until the pitman arm leaves the down position.

Req 4 If the pitman arm is moved up for at least 0.5s, right blinking shall be active until the pitman arm leaves the up position.

The final requirement concerns safety:

* This work has been partially funded by the German Federal Ministry of Education and Research (BMBF) under research grant 01IS15031H (ASSUME).

Req 5 Left and right blinking must not be active together.

State of the art tool suites for embedded systems development, such as BTC EmbeddedPlatform[1], allow generating and executing test cases directly from requirements. For this it is necessary to formalize the textual requirements, by expressing them in a formal language that is understood by the testing tool. This paper focuses on the formal language proposed by the the EmbeddedPlatform, the Simplified Universal Pattern (SUP). Pattern languages, such as SUP, are easier to use for engineers than temporal logics such as LTL coming from theoretical computer science. Pattern languages provide a limited set of templates (patterns) with a fixed and well defined semantic. For formalizing a requirement, the engineer picks a pattern and fills in the pattern parameters with simple expressions describing states and events.

Since the requirements form the basis for designing models and test cases, it is important to have high quality requirements. One quality indicator is consistency of requirements [25]. Informally, a set of requirements is consistent if it is free of contradictions. Having formalized requirements enables reasoning about consistency of requirements. Of course, this requires a formal definition of requirement consistency itself. This paper introduces a new formal definition called *partial consistency*. Partial consistency focuses on requirements expressing a *trigger/action* relationship, where the action shall occur in response to the trigger. E.g. in **Req 1** above, “the pitman arm is moved down for less than 0.5s” is the trigger and “left blinking shall be active for 3s” is the action. Checking parial consistency consists of two parts. Firstly, the set of requirements is divided into one set consisting of all the requirements where the action is a response to the trigger (such in requirements **Req 1** to **Req 4** above) and another set containing all the other requirements (**Req 5** in the example above, which does not have a trigger). Secondly, the requirements in the first set are analyzed pairwise for critical cases where timing of the triggers causes a conflict. Here, a conflict means that two actions have to occur at the same time (because of the timing of the triggers), but either contradict each other or violate (together) the requirements in the second set. In the example requirements above, **Req 1** and **Req 2** cause together a violation of **Req 5**: If the pitman arm is moved up and down shortly in sequence, either both right and left blinking is active at the same time, which violates **Req 5**, or one of **Req 1** or **Req 2** is violated. Note that partial consistency does not forbid that a system variable is affected by two actions at the same time as long both actions are satisfied. For example **Req 1** and **Req 3** are partially consistent although it is possible to trigger normal blinking while tip blinking is still active.

The rest of the paper is organized as follows: In Section 2 the SUP notation is completely described. It is compared to other pattern languages in Section 3 by sketching a mapping between SUP and SPS, a popular pattern system proposed by Dwyer et. al. [10] and extended by Konrad and Cheng [20]. In Section 4 existing definitions of consistency are presented followed by *partial consistency* in Section 5. A prototype implementation is described in Section 6 and evaluated in Section 7. The paper closes with a brief outlook on future work in Section 8.

[1] https://www.btc-es.de/

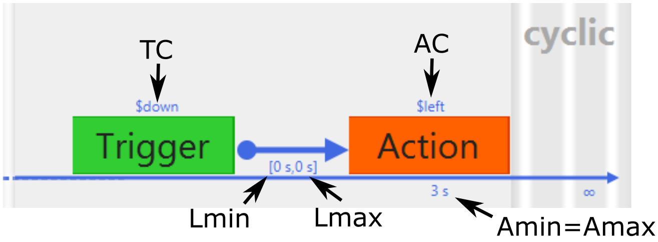


Figure 1: SUP in the EmbeddedPlatform

2 Simplified Universal Pattern

The SUP has been designed for use in BTC EmbeddedPlatform to overcome the difficulties for engineers to learn the syntax and semantic of a complex formal language, while providing a formalism that is expressive enough to formalize functional requirements for real-world embedded systems [8, 6]. The SUP can be seen as a template for formal requirements with gaps, called parameters, that are filled by the engineer with side-effect free expressions over the system variables. In the EmbeddedPlatform, a rich C-like expression language has been chosen. In our prototype implementation, we have adopted this expression language but restrict ourselves to boolean combination of linear constraints (over integer and real variables). The EmbeddedPlatform provides a graphical editor together with a methodology to guide the engineer through the formalization process.

2.1 Syntax and Semantics

The SUP assumes discrete time with a fixed step size. An instance of the SUP can be seen as some kind of state machine that consumes a trace (see Definition 1) of the system under development and decides if the SUP instance is satisfied or violated.

This paper uses a mathematical notation of the SUP that is inspired by the graphical representation in the EmbeddedPlatform given in Figure 1. In fact it is an extension of the notation found in [8]. We denote SUP instances by

$$(TSE,TC,TEE)[Tmin,Tmax] \xrightarrow{[Lmin,Lmax]} (ASE,AC,AEE)[Amin,Amax].$$

An SUP instance consists of two parts, the *trigger* $(TSE,TC,TEE)[Tmin,Tmax]$ and the *action* $(ASE,AC,AEE)[Amin,Amax]$. Both trigger and action describe behavior of the system, and the action shall occur at the same time or after the trigger. Both trigger and action have a start and end condition (called events) and optionally a minimum and maximum duration. The time between trigger and action is called *local scope* and given as an interval $[Lmin,Lmax]$. All together the SUP has 15 parameters that are summarized in Table 1. The trigger exit condition, action exit condition and the maxTime parameters are not used in this paper and therefore not part of the above notation. Most of the parameters have a default value (third column in Table 1), e.g. $TSE = TEE = TC$ which allows to omit the TSE and TEE parameters in the notation (see Section 2.2). Most time parameters may be set to infinity (last column) to indicate an open time bound.

Parameter	Abbrev.	Type	Default	Inf
Trigger Start Event	TSE	Bool	TC	–
Trigger End Event	TEE	Bool	TC	–
Trigger Condition	TC	Bool	*true*	–
Trigger Exit Condition	TEC	Bool	*false*	–
Trigger Duration Min	Tmin	Time	0	✓
Trigger Duration Max	Tmax	Time	0	✓
Local Scope Min	Lmin	Time	0	✗
Local Scope Max	Lmax	Time	0	✓
Action Start Event	ASE	Bool	AC	–
Action End Event	AEE	Bool	AC	–
Action Condition	AC	Bool	–	–
Action Exit Condition	AEC	Bool	*false*	–
Action Duration Min	Amin	Time	0	✓
Action Duration Max	Amax	Time	0	✓
Global Scope	maxTime	Time	∞	✓

Table 1: SUP parameters

Besides the parameters, the semantics of an SUP instance is controlled by its *interpretation*, *activation mode* and *startup phase*. There are three different interpretations for the SUP that define the relation of trigger and action:

progress If the trigger occurs, the action must occur in response.

invariant If the trigger occurs, the action must occur in the same step.

ordering If the action occurs, the trigger must have occurred before.

One pair of trigger and action is called an observation cycle of the SUP. Because the SUP instance is an automaton, after the trigger has been observed further occurrences of the trigger are ignored until the action occurs. The activation mode defines whether there is one or multiple observation cycles. There are three activation modes:

cyclic If an observation cycle is finished, a new observation cycle starts.

first After the first complete observation cycle, the SUP instance is satisfied; no second observation cycle starts.

init Same as *first*, but the trigger must occur at the start of the observation cycle.

The *startup phase* determines the start of the first observation cycle of the SUP. There are three startup phases defined:

immediate The observation cycle starts immediately, i.e. in step 0.

after N steps The observation cycle starts in step N, where N is a parameter of type Time.

after reaching R The observation cycle starts in step following the first occurrence of R, where R is a condition.

The arrow notation introduced above is for the *invariant* and *progress* interpretation with startup phase *immediate* and activation mode *cyclic*. For the *ordering* interpretation the right-pointing arrow ($\rightarrow$) in the notation is replaced with the reverse arrow ($\leftarrow$). To denote that one of the activation modes *first* or *init* is used instead of the default *cyclic* activation mode, the SUP instance is prefixed with a keyword `first` resp. `init`.

In the following, the details of the SUP semantics for the *progress* interpretation are described, followed by a more detailed description of the *ordering* interpretation. The *invariant* interpretation is a special case of the *progress* interpretation where all parameters except TC, AC and maxTime are set to default values. Note that this is a description of the semantics implemented in the consistency analysis prototype.

Trigger The trigger starts with the first observation of the *Trigger Start Event (TSE)*. In activation mode *initial*, the TSE must occur exactly at the end of the startup phase. The trigger phase ends with the first observation of the *trigger end event (TEE)* in the interval $t_{TSE} + Tmin \leq t_{TEE} \leq t_{TSE} + Tmax$, where t_{TSE}, t_{TEE} are the time points at which the events are observed. Between TSE and TEE, i.e. at $t_{TSE} < t < t_{TEE}$ the *Trigger Condition (TC)* must hold. If this interval exceeds without TEE being observed, or the TC is violated, the observation cycle is aborted. In activation modes *cyclic* and *first*, a new activation cycle starts by awaiting the next TSE.

Action The *action* consisting of *Action Start Event (ASE)*, *Action Condition (AC)*, *Action End Event (AEE)*, and *Action Duration* $[Amin, Amax]$ works analogous to the trigger, except that the observer stops with the *failure* signal on violation of the AC or the bounds. If the AEE is successfully observed within the specified interval, the observation cycle ends successfully (without emitting *failure*). In activation mode *cyclic*, a new observation cycle is started; the TSE of the next cycle may occur together with the last cycle's AEE.

Local Scope The *Local Scope* is a time interval $[Lmin, Lmax]$ describing the time window for the ASE, i.e. the first occurence t_{ASE} of ASE after TEE must be located in the interval $t_{TEE} + Lmin \leq t_{ASE} \leq t_{TEE} + Lmax$. Otherwise, the observer emits *failure* and stops.

The following special cases have to be considered:

Infinite Time Bounds All time bounds except Lmin and the parameter N from the startup phase *after N steps* may be set to infinity. This means the following: Setting $Lmax = \infty$ means that the ASE may occur any time (but at least $Lmin$ steps after TEE) in the future. Setting $Amin = \infty$ means that the AC must be true forever after ASE is observed, except the observation cycle is canceled by the AEC.

In the interpretation *ordering*, the local scope has only a lower bound and the upper bound is always infinite. The above behavior is modified as follows: If the ASE occurs together with the start of the trigger or earlier than $Lmin$ steps after complete observation of the trigger, i.e. on $t_{ASE} < t_{TEE} + Lmin$, the observer emits *failure*. If the AEE occurs too early or too late or the AC is violated, the observation does not fail but the observer waits for another ASE.

2.2 Examples

In the remaining of the paper we omit parameters with default values in the SUP notation. Intervals $[\ell,\ell]$ are abbreviated to $[\ell]$. For example $p \rightarrow q$ is a shorthand for

$$(p,p,p)[0,0] \xrightarrow{[0,0]} (q,q,q)[0,0]$$

and $p[5] \rightarrow (q,r,s)[0,\infty)$ abbreviates

$$(p,p,p)[5,5] \xrightarrow{[0,0]} (q,r,s)[0,\infty).$$

In the following, we formalize the requirements from the introduction. We model the pitman arm's positions as two predicates *up*, *down* and left/right blinking as *left* resp. *right*. We use a fixed step size of $0.1s$.

Req 1 If the pitman arm is moved down for less than 0.5s, left blinking shall be active for 3s.

$$(down, down, \neg down)[0,5] \rightarrow left[30]$$

Note that the trigger duration includes the step where the TEE ($\neg down$) occurs. So in fact a strict upper bound for the duration of *down* is encoded.

Req 2 Right tip blinking analogous: $(up, up, \neg up)[0,5] \rightarrow right[30]$

Req 3 If the pitman arm is moved down for at least 0.5s, left blinking shall be active until the pitman arm leaves the down position.

$$down[5] \rightarrow (left, left, \neg down)[0,\infty)$$

Req 4 Right blinking analogous: $up[5] \rightarrow (right, right, \neg up)[0,\infty)$

Req 5 Left and right blinking must not be active together.

$$true \rightarrow \neg(left \wedge right)$$

2.3 Observer Automata

In the EmbeddedPlatform, formal requirements are translated into *observers*. Observers monitor executions of a system and indicate if a requirement is violated. System executions are modeled as traces over system variables (i.e. input, output and internal variables of the system). A trace assigns a value to every variable and time step.

Definition 1 (Trace) A finite (infinite) *trace* over variables $\mathbb{X}$ is a finite (infinite) sequence $\sigma = \sigma_0\sigma_1\sigma_2\ldots$ where $\sigma_i : \mathbb{X} \rightarrow \mathscr{V}$ assigns a value $\sigma_i(x) \in \mathscr{V}_{type(X)}$ to every variable $x \in \mathbb{X}$ in step i.

We denote the set of all infinite traces over $\mathbb{X}$ by $\mathscr{T}(\mathbb{X})$, and the set of finite traces of length k by $\mathscr{T}_k(\mathbb{X})$. For $\mathbb{Y} \subseteq \mathbb{X}$ we denote by $\sigma\downarrow_{\mathbb{Y}}$ the restriction of σ to variables in $\mathbb{Y}$ and by $\sigma\downarrow_n = \sigma_0\sigma_1\ldots\sigma_{n-1}$ the prefix of length $n \in \mathbb{N}$ of σ.

In this paper, observers are modeled as Büchi automata with counters and a special failure state, called counter automata[2] [11], that accept traces over a subset of the system variables, called the observed variables of the automaton in the following. Counter automata consist of finite sets of states, transitions and counter variables that are distinct from the observed variables. Every transition is labeled with a boolean expression over observed variables and counters, called guard. In every step, a transition with satisfied guard is taken and counters are either incremented, set to some integer constant, or left unchanged.

Definition 2 (Counter Automaton) A counter automaton over a set $\mathbb{X}$ of variables is a tuple $\mathscr{A} = \langle S, \mathbb{X}, \mathbb{W}, s_0, s_f, F, T\rangle$ with states S, integer counter variables $\mathbb{W}$ (disjoint from $\mathbb{X}$), initial and failure state $s_0, s_f \in S$, a set $F \subseteq S$ of fair states such that $s_f \notin F$, and a set T of transitions. A transition $\langle s, g, \gamma, s'\rangle \in T$ consists of source and target states $s, s' \in S$, a guard $g \in Expr_{\mathbb{B}}(\mathbb{X} \cup \mathbb{W})$ (a boolean expression over $\mathbb{X} \cup \mathbb{W}$) and a function $\gamma : \mathbb{W} \to \mathbb{N} \cup \{INC, STABLE\}$.

A trace σ over $\mathbb{X} \cup \mathbb{W} \cup \{\mathtt{s}\}$ with $\mathscr{V}_{type(\mathtt{s})} = S$ is a *run* of $\mathscr{A}$ if $\sigma_0(\mathtt{s}) = s_0$, $\sigma_0(c) = 0$ for $c \in \mathbb{W}$ and for all $i \in \mathbb{N}$ ($i < n-1$ for finite traces of length n) exists $\langle s, g, \gamma, s'\rangle \in T$ such that $\sigma_i(\mathtt{s}) = s$, $\sigma_{i+1}(\mathtt{s}) = s'$, $\sigma_i \models g$ and $\sigma_{i+1}(c) = \begin{cases} \gamma(c) & \text{if } \gamma(c) \in \mathbb{N} \\ \sigma_i(c) + 1 & \text{if } \gamma(c) = INC \\ \sigma_i(c) & \text{if } \gamma(c) = STABLE \end{cases}$ for $c \in \mathbb{W}$. A run is *weak accepting* if $\sigma_i(\mathtt{s}) \neq s_f$ for all $i \in \mathbb{N}$ ($i < n$ for finite runs of length n). A run is *accepting* if it is weak accepting and $\sigma_i(\mathtt{s}) \in F$ for infinitely many steps $i \in \mathbb{N}$ (in case of an infinite run), resp. $\sigma_i(\mathtt{s}) \in F$ for some step $i \in \mathbb{N}$ (in case of a finite run).

For both infinite and finite runs $\sigma \in \mathscr{T}(\mathbb{Y})$ resp. $\sigma \in \mathscr{T}_n(\mathbb{Y})$ with $\mathbb{Y} \supseteq \mathbb{X}$ we write $\sigma \models \mathscr{A}$ if there exists an accepting run σ' of $\mathscr{A}$ with $\sigma'\downarrow_{\mathbb{X}} = \sigma\downarrow_{\mathbb{X}}$. Analogously we write $\sigma \models_w \mathscr{A}$ if there exists a weak accepting run σ' of $\mathscr{A}$ with $\sigma'\downarrow_{\mathbb{X}} = \sigma\downarrow_{\mathbb{X}}$.

Here, $\sigma_i \models g$ indicates satisfaction of guard g in step i of σ. We assume that the set $Expr_{\mathbb{B}}(\mathbb{X} \cup \mathbb{W})$ allows boolean combination of linear inequalities. Throughout this paper, we require counter automata to be *deterministic* and *complete*, i.e. in every step exactly one transition can be taken. Furthermore, *the failure state is a sink*, meaning, once entered, the failure state is never left (technically the failure state has exactly one outgoing transition that is a self loop with the guard *true*).

Note that the above definition extends counter automata with fair states, compared with the original definition in [11]. What is called acceptance in [11] is called weak acceptance here. Fair states are necessary to model instances of the SUP with parameters *Lmax* or *Amax* set to infinity, because these describe unbounded liveness properties where violation cannot be detected in finite prefixes.

[2] Because transitions are labeled with expressions that also contain system variables, the term *extended counter automata* might be more precise. We use the shorter term *counter automaton* since it is used for the same type of automata in previous work [11].

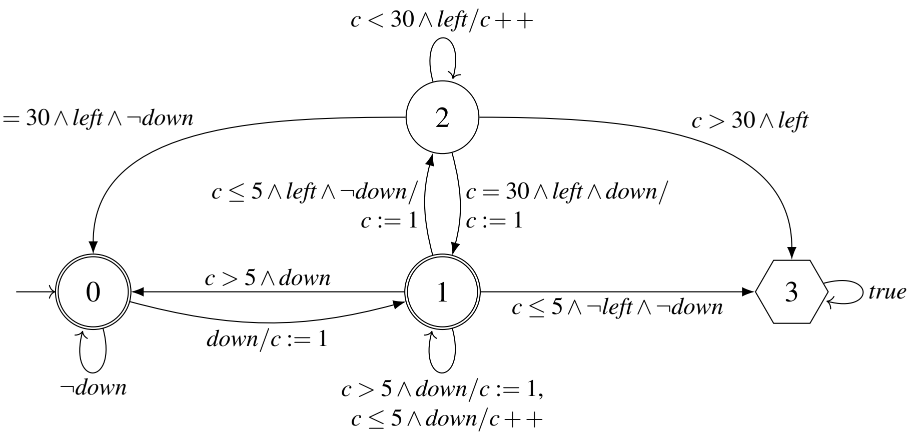

$c < 30 \wedge left/c{+}{+}$
$c = 30 \wedge left \wedge \neg down$
2
$c > 30 \wedge left$
$c \leq 5 \wedge left \wedge \neg down/$
$c := 1$
$c = 30 \wedge left \wedge down/$
$c := 1$
0
$c > 5 \wedge down$
1
3
true
$c \leq 5 \wedge \neg left \wedge \neg down$
$down/c := 1$
$\neg down$
$c > 5 \wedge down/c := 1,$
$c \leq 5 \wedge down/c{+}{+}$

Figure 2: Counter automaton for **Req 1**; state 0 is both the initial and a fair state, state 3 is the failure state; $c{+}{+}$ denotes increment of c.

Assumption 1 *For every instance R of the SUP we can construct a counter automaton $\mathscr{A}_O(R)$, called its* observer automaton*, that accepts the traces satisfying R.*

Obviously, we cannot give a formal proof for the assumption above, because we describe the SUP semantics only informally. Unfortunately there is no handy definition of the SUP semantics in temporal logics that describes the semantics of the SUP with all its parameters precisely. The formal semantics [24] of the SUP that have been kindly provided by BTC EmbeddedSystems to the author for developing his prototype, use automata networks that are very close to counter automata. Based on these formal semantics, one automata scheme for each combination of interpretation, startup phase, and activation mode of the SUP has been constructed in a computer-aided process. An automata scheme is a counter automaton over the SUP parameters instead of the system variables. For reason of space, we cannot give the construction (or a proof of its correctness) here. Unfortunately, the complete automata schemes that use all the SUP parameters are too big to be presented in a paper. Instead, Appendix A lists simplified versions of some of the automata schemes that cover the subset of the SUP used in this work. For any SUP instance R, the observer automaton $\mathscr{A}_O(R)$ can be derived from one of the automata schemes by substituting the parameters in the guards. Afterwards the following simplification steps are applied to the observer automata: (1) simplification of the guards, (2) removal of transitions with unsatisfiable guards, unreachable states and unused counters, (3) merge of equivalent states.

Example 1 *Figure 2 shows an observer automaton for* **Req 1***. It starts in state* 0*. If the trigger starts (i.e. left is true) it switches to state* 1*. If left remains true for* 5 *steps, it proceeds to state* 2*, otherwise returns to state* 0*. In state* 2*, the action is checked. If left is true for* 30 *steps, the automaton returns to the initial state, otherwise it enters the failure state. This also explains what is meant by* iterative *semantics of the SUP: Once state* 2 *is entered, the trigger variable left is not observed any more until the current action phase is violated or completed.*

3 Other pattern languages

Since the SUP as a pattern language has limitted expressiveness compared to more complex formal languages, we try in this section to give some evidence that the SUP is expressive enough to model industrial use cases. In a case study [23] performed at BOSCH the Specification Pattern System (SPS) originally proposed by Dwyer et. al. [10] and extended by Konrad and Cheng [20] has been evaluated. SPS provides a set of natural language patterns for specifying behavioral requirements with a formal semantic given in Modular Temporal Logic (MTL) [3]. In the case study, 193 out of 245 requirements could be formalized within the SPS by Konrad and Cheng. In the following we will demonstrate the usability of the SUP by presenting a mapping from the SPS patterns used in the case study to the SUP.

Definition 3 (MTL) MTL formula over a set $\mathbb{X}$ of variables are inductively defined as follows:

- A predicate over $\mathbb{X}$ is an MTL formula.
- If ϕ is an MTL formula so is $\mathbf{X}\phi$.
- If ϕ_1, ϕ_2 are MTL formula, $c \in \mathbb{T}$ and $\sim \in \{<, \leq, =, \geq, >\}$ so is $\phi_1 \mathbf{U}_{\sim c} \phi_2$.
- Boolean combinations of MTL formulas are MTL formulas.

Satisfaction of MTL is defined for traces $\sigma \in \mathscr{T}(\mathbb{X})$ and $i \in \mathbb{N}$ as follows:

- $\sigma, i \models p$ if p is a predicate over $\mathbb{X}$ and $\sigma_i \models p$.
- $\sigma, i \models \mathbf{X}\phi$ if $\sigma, i+1 \models \phi$.
- $\sigma, i \models \phi_1 \mathbf{U}_{\sim c} \phi_2$ if exists $d \sim c$, such that $\sigma, i+d \models \phi_2$ and $\sigma, j \models \phi_1$ for all $j \in \mathbb{N}$ with $i \leq j < i+d$.
- Boolean connectives are defined as usual.

MTL defines the usual abbreviations from temporal logic: $\mathbf{F}_{\sim c}\phi :\Leftrightarrow \mathit{true}\, \mathbf{U}_{\sim c}\, \phi$, $\mathbf{G}_{\sim c}\phi :\Leftrightarrow \neg \mathbf{F}_{\sim c} \neg \phi$, $\phi_1 \mathbf{W} \phi_2 :\Leftrightarrow \phi_1 \mathbf{U} \phi_2 \vee \mathbf{G}\phi_1$.

The mapping from SPS to SUP is listed in Table 2. Note that there is no general mapping from SUP to MTL and the contents of Table 2 have not been proven formally. Although the SUP instances in the table match the MTL semantics of SPS very closely, the main purpose of the table is to give evidence that the SUP may be used as a replacement of SPS in a formalization process. The SUP for "Precedence Chain 2:1" uses the *last* operator, that gives access to the value of P in the previous step. It is supported both in the EmbeddedPlatform as well as our analysis prototype. The MTL semantics of time-constrained SPS patterns has been taken from [4], for untimed patterns the original LTL semantic [10] is used. The main difference between MTL semantics of SPS and the SUP is that in the SUP there are no overlapping observation cycles (for one instance). After the trigger occurred, the SUP observer waits for the action belonging to the trigger and ignores all other triggers that may occur in between. For the SPS patterns in Table 2, a mapping is possible because (except for "Minimum duration" and "Bounded invariance") they

Pattern Name	MTL semantics	SUP
Absence	$\mathsf{G}\neg P$	$\mathit{true} \rightarrow (\neg P)$
Universality	$\mathsf{G}P$	$\mathit{true} \rightarrow P$
Existence	$\mathsf{F}P$	$\texttt{init}\ \mathit{true} \xrightarrow{[0,\infty)} P$
Response	$\mathsf{G}(P \Rightarrow \mathsf{F}Q)$	$(P \wedge \neg Q) \xrightarrow{[0,\infty)} Q$
Precedence	$(\neg P)\ \mathsf{W}\ Q$	$\texttt{first}\ Q \xleftarrow{[0,\infty)} (P \wedge \neg Q)$
Minimum duration	$\mathsf{G}(P \vee (\neg P\ \mathsf{W}\ \mathsf{G}_{\leq d}P))$	$(\neg P, P, P)[0,\infty) \rightarrow P[d]$
Maximum duration	$\mathsf{G}(P \vee (\neg P\ \mathsf{W}\ (P \wedge \mathsf{F}_{\leq d}\neg P)))$	$(\neg P, P, P)[0,\infty) \xrightarrow{[0,d]} (\neg P)$
Periodic category	$\mathsf{GF}_{\leq d}P$	$\neg P \xrightarrow{[0,d]} P$
Bounded response	$\mathsf{G}(P \Rightarrow \mathsf{F}_{\leq d}Q)$	$(P \wedge \neg Q) \xrightarrow{[0,d]} Q$
Bounded invariance	$\mathsf{G}(P \Rightarrow \mathsf{G}_{\leq d}Q)$	$(P, \neg P, \neg Q)[0,d] \rightarrow \mathit{false}$
Precedence Chain 2:1	$\neg P\ \mathsf{W}\ (S \wedge \neg P \wedge \mathsf{X}(\neg P\ \mathsf{W}\ T))$	$\texttt{first}\ (S, \mathit{true}, T)[1,\infty) \xleftarrow{[0,\infty)} \mathit{last}(P)$

Table 2: Expressing SPS patterns in SUP

have only upper time bounds, so only the largest distance from a trigger to the next action shall be observed by the automaton. The mapping for the Minimum duration pattern works because trigger and action are exclusive. For the "Bounded invariance" pattern it is possible to construct an SUP instance, where the observer is reset every time $\neg P \wedge Q$ occurs. Note that the trigger for "Bounded invariance" is the opposite of the SPS pattern semantics and the action is unsatisfiable. So, if the SPS pattern is violated, the observer for the SUP instance enters its failure state. The observer automata for the SUP instances in Table 2 are listed in Appendix A.

In SPS, a requirement has some optional scope, meaning the requirement applies only after some event P, until some event Q, or between P and Q. The semantics in Table 2 is for the global scope, meaning without restriction by events. The global scope and the scope "after P" directly correspond to startup phases "immediate" and "after reaching R" of the SUP. Using the mapping from Table 2, it shall be possible to formalize 90% of the SPS requirements from the BOSCH case study with the SUP. The remaining 10% use one of the SPS scopes "between P and Q" and "until Q", that do not have SUP counterparts.[3]

4 Existing Consistency Notions

Before we present partial consistency, we give a short summary of related work on consistency of formal requirements.

One of the earliest attempts for specifying formal requirements is Software Cost Reduction (SCR) [16, 14, 13]. In SCR, a system is described as a state machine in tabular form. The consistency checker for SCR checks for completeness and determinism of the specification. In contrast to other approaches, the consistency checker for SCR performs only static checks, but no reacheability analysis [15]. Similar methods are proposed for requirements state machines [18].

[3] Please note that this is not a limitation of the automata-based approach behind the SUP – introducing these missing scopes would be a simple extension.

Other approaches define consistency more globally. The most general definition of consistency is that there exists at least one implementation for the requirements [7]. In [11] *existential consistency* is presented, where a set of requirements is called consistent, if there exists at least one trace that satisfies all requirements. In the same work, this notion is refined to *bounded existential consistency* that can be checked using bounded model checking and *triggered existential consistency* that further restricts accepted traces to those that represent the "intended" meaning of the requirements and exclude trivial behavior that is not of practical use. In a recent work [12] the authors use an encoding of TCTL [2] formulas as SMT problems to check consistency of SPS requirements. They use a notion of consistency that is comparable to existential consistency [11] sketched above: TCTL specifications $R_1,\ldots,R_n$ are consistent, if their conjunction is satisfiable, i.e. there is a timed transition system $\mathscr{M} \models R_1 \wedge \cdots \wedge R_n$. The authors claim to exclude trivial behavior, but do not explain this in detail.

Some stronger notions of consistency take into account that the system shall be able to handle every input from the environment. In [1], requirements are implemented as labeled transition systems. Consistency is then defined on the reachable states of the system. The set of consistent states is the largest subset of the system state space such that for every input exists an output such that all requirements are satisfied and the next state is again in that set. The system is consistent if this set contains the initial state. This notion of consistency honors the fact that a component does not have control over the inputs it receives. If the requirements are consistent there exists an implementation that can deal with all inputs that are allowed by the specification.

In [22] a consistency notion called *rt-consistency* is presented that requires that every finite trace that satisfies all requirements can be extended to an infinite one. Hence, if a set of requirements is rt-consistent there exists an implementation that guarantees liveness of the system. A similar notion of consistency is implemented in the STIMULUS tool [19]. STIMULUS provides a consistency analysis by simulating functional requirements, where local non-determinism is resolved by linear constraint solving. If in some step no solution exists, STIMULUS reports an inconsistency.

5 Partial Consistency

In the following we present the main contribution of this paper: A new consistency notion for SUP called *partial consistency*. Partial consistency is based on triggered existential consistency [11]. Our experience shows that existential consistency is not enough in practice, since it does not take combinations of triggers into account. As an example, consider requirements **Req 1**, **Req 2** and **Req 5** from Section 2.2: They are existentially consistent, since, when requesting right and left tip blinking with a delay of three seconds, all requirements are satisfied in one run. But if the delay is shorter, one requirement is violated. Inconsistencies of this kind are not found by existential consistency. The user expects that a consistent set of requirement does not restrict the inputs of the system, meaning that in every state every input has at least one possible output without violating a requirement. This is very close to the definition of consistency in [1] or strong consistency defined in [5].

In the following we denote, for some set $\mathscr{R}$ of SUP instances over system variables $\mathbb{X}$, the set of satisfied traces by $\mathscr{T}(\mathscr{R}) = \{\sigma \in \mathscr{T}(\mathbb{X}) \mid \sigma \models \mathscr{A}_O(R) \text{ for any } R \in \mathscr{R}\}$. The terms requirement and SUP instance are used as synonyms.

Definition 4 (Strong Consistency[5]) A set $\mathscr{R}$ of requirements is *strongly consistent* wrt. the system inputs $IN \subseteq \mathbb{X}$ if there exists a non-empty set $\Sigma \subseteq \mathscr{T}(\mathscr{R})$ such that

$$\forall \sigma \in \Sigma, \tau \in \mathscr{T}(\mathbb{X}), n \in \mathbb{N} : \sigma{\downarrow_n} = \tau{\downarrow_n} \Rightarrow \exists \sigma' \in \Sigma : \sigma'{\downarrow_n} = \sigma{\downarrow_n} \wedge \sigma'{\downarrow_{IN}} = \tau{\downarrow_{IN}}.$$

However, as seen in [1] a check for this form of consistency requires some kind of quantifier nesting in the analysis. Although current solvers have limited support for quantifiers, we consider using SMT with nested quantifiers impractical for real-world examples. As a consequence, we try not to consider all possible input traces, but try to characterize and check those ones that may cause conflicts. Furthermore we don't want to distinguish explicitly between inputs and outputs of the system. The reason for the latter is that partial consistency shall also be applicable to *semi-formal* requirements. Semi-formal requirements use so-called *macros* [17] instead of the system variables. In a second development step, macros are mapped to expressions over the system variables. This allows reasoning on requirements before the system interface is fully specified. Since a macro can depend on both input and output variables, it is not possible to say that a macro is an input or an output.

The *partial consistency* analysis focuses on those SUPs that we call *reactive*:

Definition 5 An instance of the SUP is *reactive* if it has interpretation `invariant` or `progress` but is not of the form $true \xrightarrow{[0,0]} P$.

All other requirements are called *invariant* in the following. SUP instances with the *ordering* interpretation are not part of the reactive requirements because the action is not required to occur in response to the trigger. We designed the partial consistency analysis with two observations in mind:

1. The system cannot influence when the triggers of reactive SUPs occur, since they depend usually on inputs.

2. Conflicts are most likely caused by contradicting actions, that are forced (by occurrence of the triggers) to occur at the same time.

The strategy for consistency analysis is to inspect the *Lmin*, *Lmax* and *Amin* parameters of two reactive SUP instances and to calculate a critical delay between the triggers of the two requirements that may cause a conflict. We call this *partial consistency* because this strategy will of course only discover conflicts between pairs of requirements. Note that with the *ordering* interpretation, the action is not required to occur finally after the trigger. For two SUP instances R_1 and R_2 the actions must occur at the same time if, for the time points t_1 and t_2 marking completion of the trigger phases,

$$\forall l_1 \in LS_{R_1} \forall a_1 \in AD_{R_1} \forall l_2 \in LS_{R_2} \forall a_2 \in AD_{R_2} : \\ [t_1 + l_1, t_1 + l_1 + a_1] \cap [t_2 + l_2, t_2 + l_2 + a_2] \neq \emptyset \quad (1)$$

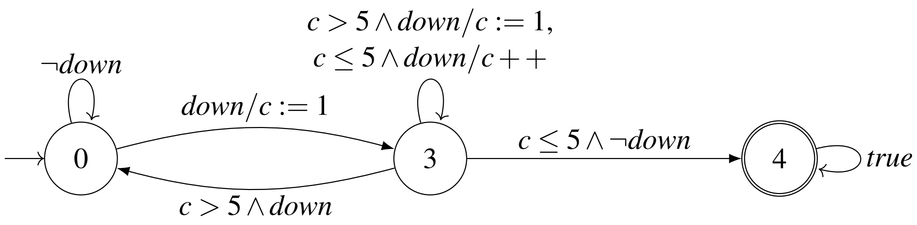

$c > 5 \wedge down / c := 1,$
$c \leq 5 \wedge down / c{+}{+}$
$\neg down$
$down / c := 1$
$c \leq 5 \wedge \neg down$
$true$
$c > 5 \wedge down$
0
3
4

Figure 3: Trigger automaton for **Req 1**

where $LS_R = [Lmin_R, Lmax_R]$ (resp. $LS_R = [Lmin_R, \infty)$ if $Lmax_R = \infty$) is the local scope and $AD_R = [Amin_R, Amax_R]$ (resp. $AD_R = [Amin_R, \infty)$ if $Amax_R = \infty$) is the action duration of an SUP instance $R \in \{R_1, R_2\}$. Eliminating the quantifiers leads to

$$Lmax_{R_2} - Lmin_{R_1} - Amin_{R_1} \leq t_1 - t_2 \leq Lmin_{R_2} + Amin_{R_2} - Lmax_{R_1}. \tag{2}$$

if all of $Lmin_{R_1}, Amin_{R_1}, Lmin_{R_2}, Amin_{R_2} \neq \infty$. If one of the parameters is set to ∞, Equation (1) is still valid, but the result of quantifier elimination changes: If $Amin_{R_1} = \infty$ then the left inequality of Equation (2) collapses to *true*, and analogously the right inequality becomes *true* if $Amin_{R_2} = \infty$. If $Lmin_{R_1} = \infty$ and $Amin_{R_2} \neq \infty$, or $Lmin_{R_2} = \infty$ and $Amin_{R_1} \neq \infty$, then Equation (1) is unsatisfiable and Equation (2) becomes *false*.

Example 2 *For requirements **Req 1** and **Req 2** (see Section 2.2) this leads to*

$$0 - 0 - 30 = -30 \leq t_1 - t_2 \leq 0 + 30 - 0 = 30$$

meaning that right and left blinking overlaps if both right and left tip blinking is requested within 30 steps ($= 3s$).

In the following, we denote the property stated in equation (2) as $\texttt{Interfere}_{(R_1,R_2)}(t_1,t_2)$. We assume that the *Lmin*, *Lmax* and *Amin* parameters are constants[4]. We follow the approach of [11] to construct for some requirement R over system variables $\mathbb{X}$ an observer automaton $\mathscr{A}_O(R)$ over $\mathbb{X}$ (the counter automaton as described in 1), and a second counter automaton $\mathscr{A}_T(R)$ (over some subset $\mathbb{Y} \subseteq \mathbb{X}$) called its *trigger automaton*. Trigger automata are constructed in a way such that their fair state is entered when the trigger phase of R is complete. One can think of the trigger automaton being the observer automaton for R where the action has been replaced by *true*. The failure state of a trigger automaton is not reachable.

Example 3 *The trigger automaton for **Req 1** is depicted in Figure 3.*

Definition 6 (Partial Consistency) For requirements R_1, R_2 and R, trace σ and $k \in \mathbb{N}$ ($k < n - 1$

[4] Note that the EmbeddedPlatform allows expressions over system variables here.

if σ is finite with length n) define

$$\begin{aligned}\texttt{TriggerAt}_R(\sigma,i) &:\Leftrightarrow \sigma{\downarrow_i} \not\models \mathscr{A}_T(R) \wedge \sigma{\downarrow_{i+1}} \models \mathscr{A}_T(R)\\ \texttt{Cond}_{\{R_1,R_2\},k}(\sigma) &:\Leftrightarrow \exists i,j\in\mathbb{N} : i \le k \wedge j \le k \wedge \texttt{TriggerAt}_{R_1}(\sigma,i)\\ &\qquad\qquad \wedge \texttt{TriggerAt}_{R_2}(\sigma,j) \wedge \texttt{Interfere}_{(R_1,R_2)}(i,j)\end{aligned}$$

Two reactive requirements R_1, R_2 and a set $\mathscr{R}_{inv}$ of invariant requirements are *partially consistent* if for all $k \in \mathbb{N}$

$$\begin{aligned}\big(\exists\sigma_1 \in \mathscr{T}(\mathscr{R}_{inv}\cup\{R_1\}) : \texttt{Cond}_{\{R_1,R_2\},k}(\sigma_1) \wedge \exists\sigma_2 \in \mathscr{T}(\mathscr{R}_{inv}\cup\{R_2\}) : \texttt{Cond}_{\{R_1,R_2\},k}(\sigma_2)\big)\\ \Rightarrow \big(\exists\sigma \in \mathscr{T}(\mathscr{R}_{inv}\cup\{R_1,R_2\}) : \texttt{Cond}_{\{R_1,R_2\},k}(\sigma)\big)\,. \qquad (3)\end{aligned}$$

In the definition, the condition $\texttt{TriggerAt}_R(\sigma,i)$ is true if the trigger of R is completed in the trace σ at step $i+1$. So $\texttt{Cond}_{\{R_1,R_2\},k}(\sigma)$ returns true if R_1 and R_2 are triggered in σ before step k such that the $\texttt{Interfere}$ condition is satisfied. We check all pairs of reactive requirements: If we can satisfy each requirement separately while triggering both requirements, can we satisfy both requirements at once? In both the premise (left-hand side of the implication sign ($\Rightarrow$) in Equation 3) and the consequence (right-hand side of the implication), traces are searched where all invariant requirements are satisfied and the delay between triggers of R_1 and R_2 is in the interval given by Equation (2).

Theorem 1 *If, for two reactive requirements R_1 and R_2, and a set $\mathscr{R}_{inv}$ of invariant requirements, the trigger automata of R_1 and R_2 depend only on input variables $IN \subseteq \mathbb{X}$ (i.e. $\mathscr{A}_T(R_1)$ is a counter automaton over some set $\mathbb{Y}_1 \subseteq IN$ and $\mathscr{A}_T(R_2)$ is a counter automaton over $\mathbb{Y}_2 \subseteq IN$), then strong consistency of $\{R_1,R_2\}\cup\mathscr{R}_{inv}$ implies partial consistency of R_1, R_2 and $\mathscr{R}_{inv}$.*

Proof. Assume that $\mathscr{R} = \{R_1,R_2\}\cup\mathscr{R}_{inv}$ is strongly consistent. We have to show that, for any $k \in \mathbb{N}$, the premise of partial consistency implies the consequence of partial consistency for the same k. Assume the premise of partial consistency holds for some $k \in \mathbb{N}$, i.e. there exists some trace $\tau \in \mathscr{T}(\mathscr{R}_{inv}\cup\{R_1\})$ such that $\texttt{Cond}_{\{R_1,R_2\},k}(\tau)$ holds. By the definition of strong consistency, there is some $\sigma' \in \mathscr{T}(\mathscr{R})$ such that $\sigma'\downarrow_{IN} = \tau\downarrow_{IN}$ (note that $\sigma\downarrow_0 = \tau\downarrow_0$ holds for arbitrary $\sigma \in \Sigma$). Remember $\mathscr{A}_T(R_1)$ is a counter automaton over some set $\mathbb{Y}_1 \subseteq IN$. For any prefix $\tau\downarrow_i$ of τ, by Definition 2, $\tau\downarrow_i \models \mathscr{A}_T(R_1)$ if, and only if, there exists some finite accepting run τ' of $\mathscr{A}_T(R_1)$ such that $\tau'\downarrow_{\mathbb{Y}_1} = \tau\downarrow_i\downarrow_{\mathbb{Y}_1}$. Because $\mathbb{Y}_1 \subseteq IN$, also $\sigma'\downarrow_i\downarrow_{\mathbb{Y}_1} = \tau'\downarrow_{\mathbb{Y}_1}$. As a consequence, $\sigma'\downarrow_i \models \mathscr{A}_T(R_1)$ if, and only if, $\tau\downarrow_i \models \mathscr{A}_T(R_1)$. The same holds for $\mathscr{A}_T(R_2)$. To sum up, $\texttt{Cond}_{\{R_1,R_2\},k}(\sigma')$ holds and, together with $\sigma' \in \mathscr{T}(\mathscr{R})$, finally R_1, R_2, and $\mathscr{R}_{inv}$ are partially consistent. □

Our implementation uses bounded model checking (BMC) [9]. BMC decides if some property can be reached within n steps in a transition system by unrolling the transition relation n times. Therefore we cannot check infinite traces. Instead we use finite acceptance criteria that can be checked with BMC and allow to infer the (non)existence of satisfying infinite traces.

Definition 7 (Finite Acceptance Criteria) For a set $\mathscr{R}$ of requirements define the following sets of satisfying traces over $\mathbb{X}$:

- *Bounded acceptance:* The set of finite traces of length $k \in \mathbb{N}$ that satisfy $\mathscr{R}$ is $\mathscr{T}_k(\mathscr{R}) := \{\sigma \in \mathscr{T}_k(\mathbb{X}) \mid \sigma \models_w \mathscr{A}_O(R) \text{ for each } R \in \mathscr{R}\}$.
- *Searching for a loop [11]:* We introduce a modified acceptance relation: For $i < k \in \mathbb{N}$ and some finite trace $\sigma \in \mathscr{T}_k(\mathbb{Y})$ ($\mathbb{Y} \supseteq \mathbb{X}$) we write $\sigma \models_{loop(i,k)} \mathscr{A}$ if there exists a finite accepting run σ' of length $k+1$ for $\mathscr{A}$ such that $\sigma{\downarrow}_{\mathbb{X}} = \sigma'{\downarrow}_{\mathbb{X}}$, $\sigma'_i = \sigma'_k$ and $\sigma'_j(\mathtt{s}) \in F$ for some $j \in \mathbb{N}$ with $i \leq j \leq k$. Here $\mathtt{s}$ is the state variable and F the set of fair states of $\mathscr{A}$. We define the set of satisfying traces with a loop from i to k as $\mathscr{T}_{(i,k)}(\mathscr{R}) := \{\sigma \in \mathscr{T}_j(\mathbb{X}) \mid \sigma \models_{loop(i,j)} \mathscr{A}_O(R) \text{ for each } R \in \mathscr{R}\}$.

Lemma 1 *For a set $\mathscr{R}$ of requirements and all $i,k \in \mathbb{N}$ with $i \leq k$*

$$\mathscr{T}_{(i,k)}(\mathscr{R}) \subseteq \{\sigma \downarrow_k \mid \sigma \in \mathscr{T}(\mathscr{R})\} \subseteq \mathscr{T}_k(\mathscr{R}).$$

proof sketch. The second inclusion $\{\sigma \downarrow_k \mid \sigma \in \mathscr{T}(\mathscr{R})\} \subseteq \mathscr{T}_k(\mathscr{R})$ follows from Definition 2. The idea behind acceptance with a loop is that the system state at start of the loop is indistinguishable from the state at the end of the loop. Every prefix with a loop can be extended to an infinite trace by infinitely repeating the loop, so $\mathscr{T}_{(i,k)}(\mathscr{R}) \subseteq \{\sigma \downarrow_k \mid \sigma \in \mathscr{T}(\mathscr{R})\}$. For details see [11]. Note that compared to [11] our counter automata have fair states. Because we require that the counter automaton is in a fair state at least once between i and j in the finite run σ', this fair state is visited infinitely often in the infinite run. □

Definition 8 (Bounded Partial Consistency) Two reactive requirements R_1, R_2 and a set $\mathscr{R}_{inv}$ of invariant requirements are (α,β)*-bounded partially consistent* (with $\alpha,\beta \in \mathbb{N}$ such that $\alpha,\beta \geq 1$) if

$$\begin{aligned}&(\exists i < \alpha \exists \sigma \in \mathscr{T}_{(i,\alpha)}(\mathscr{R}_{inv} \cup \{R_1\}) : \mathtt{Cond}_{\{R_1,R_2\},i}(\sigma))\\ &\wedge (\exists i < \alpha \exists \sigma \in \mathscr{T}_{(i,\alpha)}(\mathscr{R}_{inv} \cup \{R_2\}) : \mathtt{Cond}_{\{R_1,R_2\},i}(\sigma))\\ &\qquad\qquad \Rightarrow (\exists \sigma \in \mathscr{T}_{\alpha+\beta}(\mathscr{R}_{inv} \cup \{R_1,R_2\}) : \mathtt{Cond}_{\{R_1,R_2\},\alpha}(\sigma)).\end{aligned}$$

Analogous to partial consistency, we call the left-hand side of the implication in Definition 8 the premise, and the right-hand side the consequence of bounded partial consistency. Larger values for α and β increase the completeness of bounded partial consistency wrt. partial consistency, but also increase the length of the traces to be checked. Lower values for α may lead to the case that the premise of bounded partial consistency is not satisfiable at all and thereby no inconsistencies can be found. For example consider the observer automaton for **Req 1** given in Figure 2: A satisfying run with a loop that also satisfies $\mathtt{Cond}$ has to go through states 0, 1 and 2, which takes $Tmin + Amin = 35$ steps in sum. As a rule of thumb, for all SUP instances $\alpha \gg Tmin + Lmin + Amin$, because this is the minimum length of a complete observation cycle and therefor of a trace with a loop. The consequence of bounded partial consistency checks for the existence of a trace that satisfies all requirements, where the triggers occur before step α. If β is too small, it might be the case that the prefixes that we are checking end before some observer automaton is forced into its failure state. Because entering the failure state may be delayed up to $Lmax$ or even $Lmax + Amax$ steps (until the end of the action phase), set $\beta \gg Lmax$, or $\beta \gg Lmax + Amax$ if $Amax \neq \infty$ and $AEE \neq AC$.

Bounded partial consistency is a sound, but not complete, under-approximation of partial consistency.

Theorem 2 *For any two reactive requirements R_1 and R_2, any set $\mathscr{R}_{inv}$ of invariant requirements, and any $\alpha, \beta \in \mathbb{N}$ with $\alpha, \beta \geq 1$, partial consistency of R_1, R_2 and $\mathscr{R}$ implies (α, β)-bounded partial consistency of R_1, R_2 and $\mathscr{R}$.*

Proof. Assume two reactive requirements R_1 and R_2 and some set $\mathscr{R}_{inv}$ of invariant requirements are partially consistent. Choose some arbitrary $\alpha \in \mathbb{N}$ such that $\alpha \geq 1$. If, for the chosen α, the premise of bounded partial consistency does not hold, then the requirements are bounded partially consistent by definition. In the following consider the other case, i.e. the premise of bounded partial consistency holds for the chosen α. By Lemma 1, existence of some $\sigma_1 \in \mathscr{T}_{(i,\alpha)}(\mathscr{R}_{inv} \cup \{R_1\})$ implies existence of some $\sigma_1' \in \mathscr{T}(\mathscr{R}_{inv} \cup \{R_1\})$ such that $\sigma_1' \downarrow_\alpha = \sigma_1$, and, by definition, $\texttt{Cond}_{\{R_1,R_2\},i}(\sigma_1)$ implies $\texttt{Cond}_{\{R_1,R_2\},\alpha}(\sigma_1')$ for $i < \alpha$. The same is true for $\mathscr{R}_{inv} \cup \{R_2\}$. Hence the premise of partial consistency holds for $k = \alpha$. Assuming R_1, R_2 and $\mathscr{R}_{inv}$ are partially consistent, also the consequence of partial consistency holds for $k = \alpha$. I.e., there exists some trace $\sigma \in \mathscr{T}(\mathscr{R}_{inv} \cup \{R_1, R_2\})$ such that $\texttt{Cond}_{\{R_1,R_2\},\alpha}(\sigma)$ holds. By Lemma 1, $\sigma \downarrow_{\alpha+\beta} \in \mathscr{T}_{\alpha+\beta}(\mathscr{R}_{inv} \cup \{R_1, R_2\})$ for any $\beta \in \mathbb{N}$ with $\beta \geq 1$. Again, $\texttt{Cond}_{\{R_1,R_2\},\alpha}(\sigma)$ implies $\texttt{Cond}_{\{R_1,R_2\},\alpha}(\sigma \downarrow_{\alpha+\beta})$, so the consequence of bounded partial consistency holds as well. To sum up: R_1, R_2 and $\mathscr{R}_{inv}$ are bounded partially consistent. □

Although (bounded) partial consistency is based on (bounded) triggered existential consistency, they are incomparable. The example from the introduction is triggered existentially consistent but not partially consistent, as shown in Section 7. On the other hand, the running example from [11] is bounded partially consistent but not triggered existentially consistent. Note also that bounded triggered consistency implies triggered consistency while partial consistency implies bounded partial consistency.

6 Implementation

We implemented a prototype of the partial consistency using the general-purpose SMT solver Z3 [21] as a backend. We widely reuse the implementation from [11] and extend it with the `Interfere` condition. Checking a pair of requirements for bounded partial consistency results in three BMC problems: Two for the premise (the left-hand side of the implication sign $\Rightarrow$) and one for the consequence (the right hand side of $\Rightarrow$) in Definition 8.

A BMC problem consists of three conditions over the system state X in the current and the next step, denoted by X': The initial condition $I(X)$, the transition relation $T(X, X')$ and the target $F(X)$. To check if F can be reached within n steps we introduce $n+1$ copies $X_0, \dots, X_n$ of the system variables X and check if

$$I(X_0) \wedge \bigwedge_{i=1}^{n} T(X_{i-1}, X_i) \wedge F(X_n)$$

is satisfiable. The size of the resulting SMT problems for the pair (R_1, R_2) is in

$$\mathcal{O}\left((\alpha+\beta)\left(|\mathscr{A}_T(R_1)|+|\mathscr{A}_T(R_2)|+\sum_{R\in\mathscr{R}_{inv}\cup\{R_1,R_2\}}|\mathscr{A}_O(R)|\right)\right).$$

Here, $|\mathscr{A}|$ is the size of a counter automaton[5].

The encoding choosen in [11] introduces for every observer or trigger automaton $\mathscr{A}$ a boolean variable $\texttt{fail}_{\mathscr{A}}$ resp. $\texttt{fair}_{\mathscr{A}}$ that is *true* if the automaton is in its failure resp. a fair state in the current step. We found a finite trace satisfying some set $\mathscr{R}$ of requirements if, for each requirement $R \in \mathscr{R}$, in some step in every step $\texttt{fair}_{\mathscr{A}_O(R)} = \mathit{true}$ and $\texttt{fail}_{\mathscr{A}_O(R)} = \mathit{false}$ in all steps. Because the failure states cannot be left, it is enough to check $\texttt{fail}_{\mathscr{A}_O(R)} = \mathit{false}$ in the last step. To encode $\texttt{Trigger}_{\{R_1,R_2\},k}$, we introduce a variable t_R for $R \in \{R_1, R_2\}$ that is set to the current step index at the time $\texttt{fair}_{\mathscr{A}_T(R)}$ becomes true and remains stable in the rest of the trace. Then we encode $\texttt{Cond}_{\{R_1,R_2\},k}$ in the target as

$$\texttt{fair}_{\mathscr{A}_T(R_1)} \wedge \texttt{fair}_{\mathscr{A}_T(R_2)} \wedge t_{R_1} < k \wedge t_{R_2} < k \wedge \texttt{Interfere}_{(R_1,R_2)}(t_{R_1}, t_{R_2}).$$

We encode satisfiability with a loop by introducing a copy x_{store} for each state variable $x \in \mathit{SVars}$. Here the state variables SVars consists of, for each counter automaton, the variable s holding the current state and the counters. We let the BMC solver "guess" the beginning of the loop by setting a boolean variable *loop* to *true* in some arbitrary step. When setting *loop*, the current state is stored into the copy variables that remain stable to the end of the trace. The transition relation of the BMC problem is extended as follows (unprimed variables denote the values in the current step and primed variables values in the next step):

$$\left(\mathit{loop} \Rightarrow \mathit{loop}' \wedge \bigwedge_{s\in \mathit{SVars}} s'_{store} = s_{store}\right) \wedge \left((\neg \mathit{loop} \wedge \mathit{loop}') \Rightarrow \bigwedge_{s\in \mathit{SVars}} s'_{store} = s\right)$$

The trace has a loop if in the last step if the state variables again equal the copy variables. So we add

$$\mathit{loop} \wedge \bigwedge_{s\in \mathit{SVars}} s_{store} = s$$

to the target of the BMC problem.

7 Evaluation

We evaluate the partial consistency on the example from the introduction. Since it is physically impossible to move the pitman arm up and down at the same time, we add a further invariant requirement:

Req 6 $\mathit{true} \rightarrow \neg(\mathit{up} \wedge \mathit{down})$

[5] More precisely the size of the formula that is required to encode its transition relation for a single step, which grows with the number of counters, transitions and the size of the guards, i.e. $|\mathscr{A}| \in \mathcal{O}\left(\Sigma_{(s,g,\gamma,s')\in T}(|g|+|\mathbb{W}|)\right)$.

Set	Premise 1	Premise 2	Consequence
Req 1, Req 2	2.88s	3.49s	7.21s
Req 1, Req 3	2.78s	2.61s	2.86s
Req 1, Req 4	3.59s	2.73s	3.79s
Req 2, Req 3	5.33s	2.37s	4.79s
Req 2, Req 4	3.21s	2.07s	2.58s
Req 3, Req 4	3.14s	–	–

Table 3: Solver run times for the analysis

The analysis finds three pairs in the example that together with **Req 5** form an inconsistency: $\{\textbf{Req 1}, \textbf{Req 2}\}$, $\{\textbf{Req 1}, \textbf{Req 4}\}$, and $\{\textbf{Req 2}, \textbf{Req 3}\}$. All three cases refer to conflicts with **Req 5** that occur when requesting blinking in one direction while tip blinking in the other direction is still active. The run times of Z3 solver (divided into the two premises and the consequence for each pair) are listed in Table 3. For the pair $\{\textbf{Req 3}, \textbf{Req 4}\}$, the first premise is unsatisfiable, and therefor the BMC problems for the second premise and consequence have been skipped (indicated by a dash in Table 3). For the remaining four pairs, both premise and consequence hold. We choose as bounds $\alpha = 40$ and $\beta = 20$. The test has been run on a Windows 7 PC with an Intel Core i5-2400@3.10GHz using Z3 4.6.0 as a backend.

It turned out that the BMC unrolling depth is the crucial factor influencing performance. Therefore we repeated our experiments using *symbolic time*, meaning we contract multiple steps into one if only counter values change[6]. This allows us to get the same results as above with only 25 unrolling steps. As another evaluation, using symbolic time we are able to check consistency of the industrial case study from [6] containing 16 formalized requirements in 80s. Although the author has not been able to prove the formal soundness results from Section 5 for symbolic time yet, no false findings of the analysis have been experienced so far.

Compared to bounded existential consistency, bounded partial consistency does not produce false inconsistencies. In contrast to [11, 12], more complex inconsistencies are found. Unfortunately in [12] no performance results are given, but we assume our approach being comparable in performance. Simulation, as in the ArgoSim[7] STIMULUS tool [19] is another promising and interesting approach. For the running example of this paper, STIMULUS finds the same inconsistencies as the bounded partial consistency analysis. However, simulation-based techniques are less powerful in resolving non-determinism and therefore require another – more imperative – style of requirements.

8 Conclusion and Future Work

In this paper, partial consistency has been introduced. Partial consistency is an extension of existential consistency that checks pairs of reactive SUPs, under the assumption that they interfere with each other. Interference of SUPs means that they are triggered in a way such that their

[6] The basic idea is to introduce an explicit integer time variable t and increase counters in every step by $t_{i+1} - t_i$ instead of 1. The maximum increment is calculated from the guards.

[7] https://argosim.com/

actions must occur at the same time. We skip pairs, where interference is not possible. Partial consistency finds more true conflicts between requirements than existential consistency alone. However, there are cases that are not recognized by partial consistency, for example

- Conflicts that involve more than two reactive requirements
- Cases where the conflict is not between the actions of two requirements but between e.g. an action and the local scope. This means that one requirement causes the action of another requirement to occur too early.

It is ongoing work to make the interference relation more generic in order to analyze a wider range of conflicts. This will also allow to adopt partial consistency for other pattern languages, for example SPS or RSL [5]. In this paper we propose a mapping from SPS to SUP instead.

Bibliography

[1] Aichernig, B.K., Hörmaier, K., Lorber, F., Ničković, D., Tiran, S.: Require, test and trace IT. In: Formal Methods for Industrial Critical Systems - 20th International Workshop, FMICS 2015, Proceedings. LNCS, vol. 9128, pp. 113–127. Springer (2015)

[2] Alur, R., Courcoubetis, C., Dill, D.: Model-checking in dense real-time. Information and computation 104(1), 2–34 (1993)

[3] Alur, R., Henzinger, T.A.: Real-time logics: Complexity and expressiveness. Information and Computation 104(1), 35–77 (1993)

[4] Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. IEEE Transactions on Software Engineering 41(7), 620–638 (2015)

[5] Baumgart, A., Böde, E., Büker, M., Werner Damm, G.E., Gezgin, T., Henkler, S., Hungar, H., Josko, B., Oertel, M., Peikenkamp, T., Reinkemeier, P., Stierand, I., Weber, R.: Architecture modeling. Tech. rep., OFFIS (3 2011), http://ses.informatik.uni--oldenburg.de/download/bib/paper/OFFIS--TR2011{_}ArchitectureModeling.pdf

[6] Becker, J.S., Bertram, V., Bienmüller, T., Brockmeyer, U., Dörr, H., Peikenkamp, T., Teige, T.: Interoperable toolchain for requirements-driven model-based development. In: ERTS 2018 (2018)

[7] Benveniste, A., et al.: Contracts for system design. Foundations and Trends in Electronic Design Automation 12(2-3), 124–400 (2018)

[8] Bienmüller, T., Teige, T., Eggers, A., Stasch, M.: Modeling requirements for quantitative consistency analysis and automatic test case generation. In: FM&MDD 2016. Computing Science Technical Report Series, vol. CS-TR-1503. Newcastle University (2016)

[9] Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without bdds. In: International conference on tools and algorithms for the construction and analysis of systems. pp. 193–207. Springer (1999)

[10] Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st international conference on Software engineering. pp. 411–420. ACM (1999)

[11] Ellen, C., Sieverding, S., Hungar, H.: Detecting consistencies and inconsistencies of pattern-based functional requirements. In: Formal Methods for Industrial Critical Systems - 19th International Conference, FMICS 2014. pp. 155–169 (2014)

[12] Filipovikj, P., Rodriguez-Navas, G., Nyberg, M., Seceleanu, C.: Smt-based consistency analysis of industrial systems requirements. In: Proceedings of the Symposium on Applied Computing. pp. 1272–1279. ACM (2017)

[13] Heitmeyer, C., Archer, M., Bharadwaj, R., Jeffords, R.: Tools for constructing requirements specification: the scr toolset at the age of ten. Tech. rep., Naval Research Lab Washington DC Center for High Assurance Computing Systems (CHACS) (2005)

[14] Heitmeyer, C., Kirby, J., Labaw, B.: Tools for formal specification, verification, and validation of requirements. In: Computer Assurance, 1997. COMPASS'97. Are We Making Progress Towards Computer Assurance? Proceedings of the 12th Annual Conference on. pp. 35–47. IEEE (1997)

[15] Heitmeyer, C.L.: Formal methods for specifying, validating, and verifying requirements. Journal of Universal Computer Science 13(5), 607–618 (2007)

[16] Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G.: Automated consistency checking of requirements specifications. ACM Transactions on Software Engineering and Methodology (TOSEM) 5(3), 231–261 (1996)

[17] Holberg, H.J., Brockmeyer, U.: Computer-aided formal specification to enable a fully automated requirements-based testing process (2012), https://www.btc-es.de/assets/files/whitepapers/computer-aided-formal-specification-to-enable.pdf

[18] Jaffe, M.S., Leveson, N.G., Heimdahl, M.P.E., Melhart, B.E.: Software requirements analysis for real-time process-control systems. IEEE transactions on software engineering 17(3), 241–258 (1991)

[19] Jeannet, B., Gaucher, F.: Debugging embedded systems requirements with stimulus: an automotive case-study. In: 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016) (2016)

[20] Konrad, S., Cheng, B.H.: Real-time specification patterns. In: Proceedings of the 27th international conference on Software engineering. pp. 372–381. ACM (2005)

[21] de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008. Proceedings. pp. 337–340. Springer (2008)

[22] Post, A., Hoenicke, J., Podelski, A.: rt-inconsistency: A new property for real-time requirements. In: Fundamental Approaches to Software Engineering - 14th International Conference, FASE 2011. Proceedings. pp. 34–49 (2011)

[23] Post, A., Menzel, I., Hoenicke, J., Podelski, A.: Automotive behavioral requirements expressed in a specification pattern system: a case study at bosch. Requirements Engineering 17(1), 19–33 (2012)

[24] Teige, T.: Simplified Universal Pattern Syntax and Semantics. BTC EmbeddedSystems (06 2017), confidential

[25] Zowghi, D., Gervasi, V.: On the interplay between consistency, completeness, and correctness in requirements evolution. Information & Software Technology 45(14), 993–1009 (2003)

A SUP Counter Automata Schemes

In the following, simplified versions of some of the counter automata are listed that the consistency analysis prototype uses as the SUP semantics. Note that the prototype has been implemented independently from the BTC embedded platform. Although the counter automata have been created based on EmbeddedPlatform documentation, the SUP semantics that is used in the BTC EmbeddedPlatform may be different.

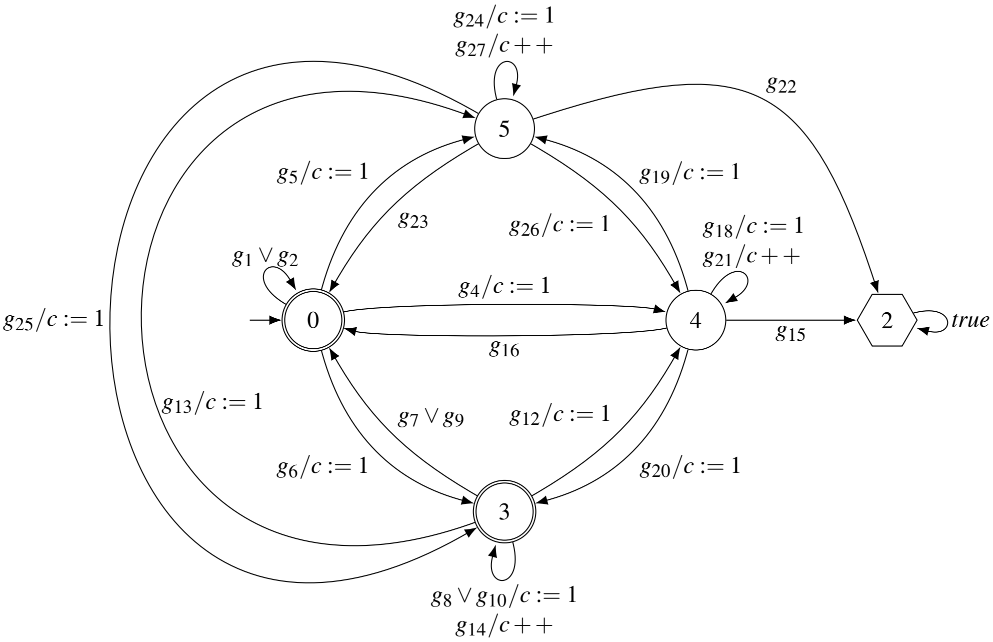


Figure 4: Counter automata scheme for interpretation *progress*, activation mode *cyclic*. If $Amin = \infty$, then state 4 is also a fair state.

A.1 cyclic-progress-immediate

With interpretation *progress*, activation mode *cyclic*, and $Lmin = 0$, the semantics of the SUP is given by the counter automata scheme in Figure 4 with following guards:

$$\begin{aligned}
g_1 &:= \neg TSE \\
g_2 &:= AEE \wedge (Amin \leq 0) \wedge TSE \wedge ASE \wedge TEE \wedge (Tmin \leq 0) \\
g_4 &:= ((Amin > 0) \vee \neg AEE) \wedge ASE \wedge TEE \wedge TSE \wedge (Tmin \leq 0) \\
g_5 &:= TEE \wedge TSE \wedge (Tmin \leq 0) \wedge \neg ASE \\
g_6 &:= ((Tmin > 0) \vee \neg TEE) \wedge TSE \\
g_7 &:= ((((Tmin > c) \vee \neg TEE) \wedge \neg TC) \vee (c > Tmax)) \wedge \neg TSE \\
g_8 &:= ((((Tmin > c) \vee \neg TEE) \wedge \neg TC) \vee ((c > Tmax) \wedge ((Tmin > 0) \vee \neg TEE))) \wedge TSE \\
g_9 &:= (((c > Tmax) \wedge TSE \wedge (Tmin \leq 0)) \vee ((Tmin \leq c) \wedge (c \leq Tmax) \wedge \neg TSE)) \\
&\quad \wedge AEE \wedge (Amin \leq 0) \wedge ASE \wedge TEE \\
g_{10} &:= AEE \wedge (Amin \leq 0) \wedge TSE \wedge (c \leq Tmax) \wedge ASE \wedge TEE \wedge (Tmin \leq c) \\
g_{12} &:= ((Tmin \leq c) \wedge (c \leq Tmax) \wedge ((Amin > 0) \vee \neg AEE) \\
&\quad \vee (TSE \wedge (Tmin \leq 0) \wedge ((Amin > 0) \vee \neg AEE))) \wedge ASE \wedge TEE \\
g_{13} &:= (((Tmin \leq c) \wedge (c \leq Tmax)) \vee (TSE \wedge (Tmin \leq 0))) \wedge TEE \wedge \neg ASE \\
g_{14} &:= ((Tmin > c) \vee \neg TEE) \wedge TC \wedge (c \leq Tmax) \\
g_{15} &:= (((Amin > c) \vee \neg AEE) \wedge \neg AC) \vee (c > Amax) \\
g_{16} &:= \neg TSE \wedge (c \leq Amax) \wedge AEE \wedge (Amin \leq c) \\
g_{18} &:= AEE \wedge TSE \wedge (c \leq Amax) \wedge ASE \wedge TEE \wedge (Amin \leq c) \wedge (Tmin \leq 0) \\
g_{19} &:= (c \leq Amax) \wedge AEE \wedge TEE \wedge (Amin \leq c) \wedge TSE \wedge (Tmin \leq 0) \wedge \neg ASE \\
g_{20} &:= ((Tmin > 0) \vee \neg TEE) \wedge (c \leq Amax) \wedge AEE \wedge (Amin \leq c) \wedge TSE \\
g_{21} &:= ((Amin > c) \vee \neg AEE) \wedge AC \wedge (c \leq Amax) \\
g_{22} &:= c > Lmax \\
g_{23} &:= \neg TSE \wedge AEE \wedge ASE \wedge (Amin \leq 0) \wedge (c \leq Lmax) \\
g_{24} &:= (AEE \wedge ASE) \wedge (Amin \leq 0) \wedge TEE \wedge TSE \wedge (c \leq Lmax) \wedge (Tmin \leq 0) \\
g_{25} &:= ((Tmin > 0) \vee \neg TEE) \wedge AEE \wedge ASE \wedge (Amin \leq 0) \wedge TSE \wedge (c \leq Lmax) \\
g_{26} &:= ((Amin > 0) \vee \neg AEE) \wedge ASE \wedge (c \leq Lmax) \\
g_{27} &:= (c \leq Lmax) \wedge \neg ASE
\end{aligned}$$

Every state corresponds to a phase of the SUP observation cycle: In state 0, the automaton waits for the TSE to occur, in state 3 it awaits the TEE, in state 5 the ASE, and finally in state 4 the AEE.

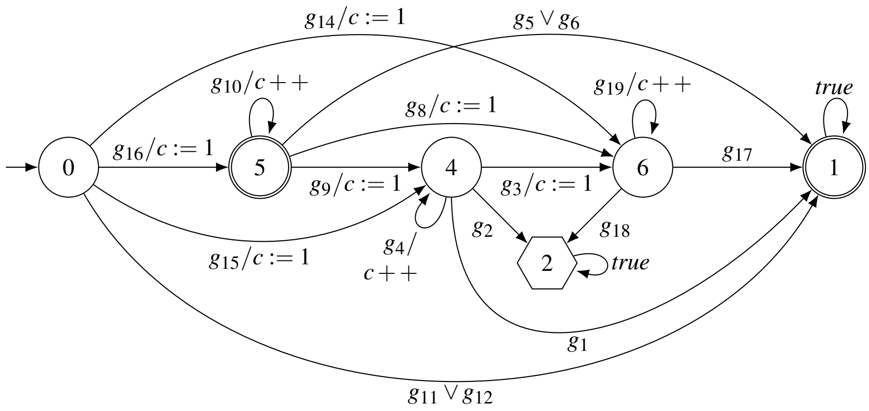


Figure 5: Counter automata scheme for interpretation *progress*, activation mode *init*. If $Amin = \infty$ then state 6 is also a fair state.

A.2 init-progress-immediate

With interpretation *progress*, activation mode *init*, and $Lmin = 0$, the semantics of the SUP is given by the counter automata scheme in Figure 5 with the following guards:

$$
\begin{aligned}
g_1 &:= AEE \wedge ASE \wedge (Amin \leq 0) \wedge (c \leq Lmax)\\
g_2 &:= c > Lmax\\
g_3 &:= ((Amin > 0) \vee \neg AEE) \wedge ASE \wedge (c \leq Lmax)\\
g_4 &:= (c \leq Lmax) \wedge \neg ASE\\
g_5 &:= (((Tmin > c) \vee \neg TEE) \wedge \neg TC) \vee (c > Tmax)\\
g_6 &:= AEE \wedge ASE \wedge (Amin \leq 0) \wedge TEE \wedge (Tmin \leq c) \wedge (c \leq Tmax)\\
g_8 &:= ((Amin > 0) \vee \neg AEE) \wedge ASE \wedge TEE \wedge (Tmin \leq c) \wedge (c \leq Tmax)\\
g_9 &:= TEE \wedge (Tmin \leq c) \wedge (c \leq Tmax) \wedge \neg ASE\\
g_{10} &:= ((Tmin > c) \vee \neg TEE) \wedge TC \wedge (c \leq Tmax)\\
g_{11} &:= \neg TSE\\
g_{12} &:= AEE \wedge (Amin \leq 0) \wedge TSE \wedge ASE \wedge TEE \wedge (Tmin \leq 0)\\
g_{14} &:= ((Amin > 0) \vee \neg AEE) \wedge ASE \wedge TEE \wedge TSE \wedge (Tmin \leq 0)\\
g_{15} &:= TEE \wedge TSE \wedge (Tmin \leq 0) \wedge \neg ASE\\
g_{16} &:= ((Tmin > 0) \vee \neg TEE) \wedge TSE\\
g_{17} &:= (c \leq Amax) \wedge AEE \wedge (Amin \leq c)\\
g_{18} &:= (((Amin > c) \vee \neg AEE) \wedge \neg AC) \vee (c > Amax)\\
g_{19} &:= ((Amin > c) \vee \neg AEE) \wedge AC \wedge (c \leq Amax)
\end{aligned}
$$

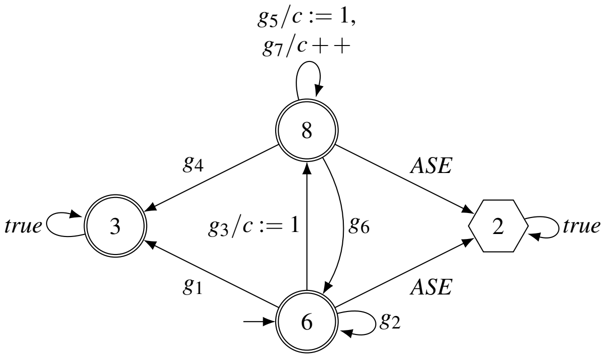


Figure 6: Counter automata scheme for interpretation *ordering*, activation mode *first*

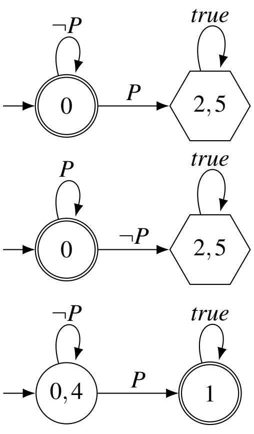
A.3 first-ordering-immediate

With interpretation *ordering*, activation mode *first*, $Lmin = 0$, and $Lmax = \infty$, the semantics of the SUP is given by the counter automata scheme in Figure 6 with following guards:

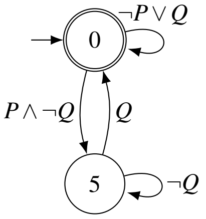
$$
\begin{aligned}
&g_1 := TEE \land TSE \land (Tmin \leq 0) \land \neg ASE\\
&g_2 := \neg TSE \land \neg ASE\\
&g_3 := (Tmin > 0) \lor \neg TEE \land TSE \land \neg ASE\\
&g_4 := (((Tmin \leq c) \land (c \leq Tmax)) \lor (TSE \land (Tmin \leq 0))) \land TEE \land \neg ASE\\
&g_5 := ((((((Tmin > c) \lor \neg TEE) \land \neg TC) \lor ((c > Tmax) \land (((Tmin > 0)) \lor \neg TEE)))\\
&\quad \land TSE \land \neg ASE\\
&g_6 := ((((((Tmin > c) \lor \neg TEE) \land \neg TC) \lor (c > Tmax)) \land \neg TSE \land \neg ASE\\
&g_7 := ((Tmin > c) \lor \neg TEE) \land TC \land (c \leq Tmax) \land \neg ASE
\end{aligned}
$$

B SUP Observer Automata for SPS Patterns

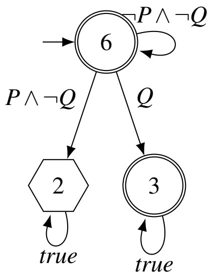
In the following, the observer automata for the SUP instances in Table 2 are listed.



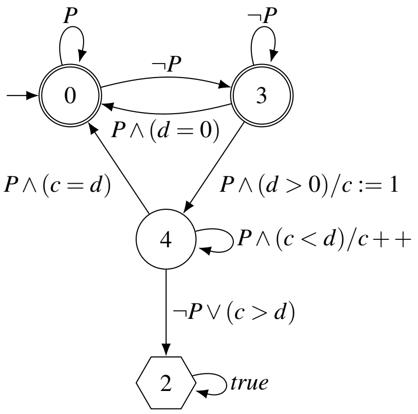
Absence (top), Universality, Existence



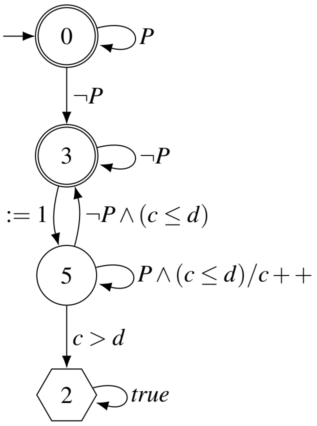
Response



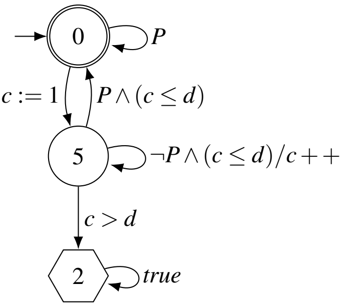
Precedence



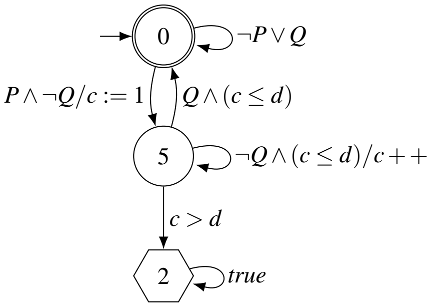
Minimum duration



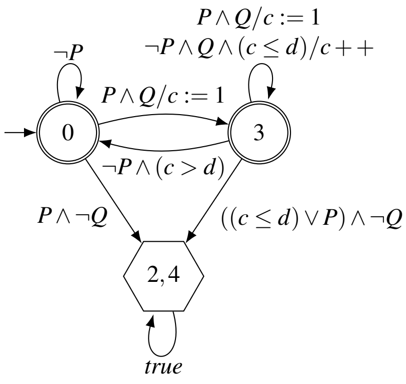
Maximum duration



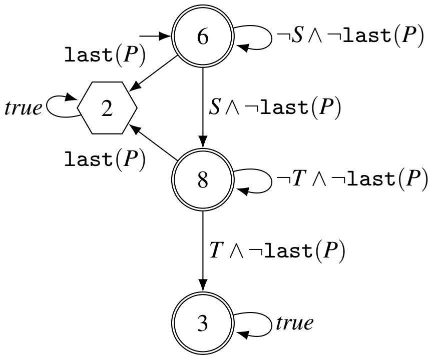
Periodic category



Bounded response



Bounded invariance



Precedence chain 2:1