Analyzing Consistency of Formal Requirements*

Jan Steffen Becker

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

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. Modern cars have a feature called tip-blinking:

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.

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

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.

And as a safety requirement:

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, meaning expressing them in a formal language that is understood by the testing tool. Following the EmbeddedPlatform's approach, the Simplified Universal Pattern (SUP) is used in this paper. Pattern languages, such as SUP, are easier to use for engineers than temporal languages 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 [24]. 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 formal requirement consistency itself. This paper introduces a new formal definition called *partial consistency*. Partial consistency focuses on requirements expressing a *trigger/action* relation ship. Here, trigger and action specify behaviors where the action shall occur in response to the trigger. The basic idea is to analyze requirements pairwise for critical cases where an 'unfortunate' timing of the triggers causes a conflict between the actions.

The rest of the paper is as follows: In Section 2 the SUP 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 [19]. 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 in Section 8.

2 Simplified Universal Pattern

The SUP has been designed for use in BTC EmbeddedPlatform to overcome the difficulties for the engineer 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 C-like expressions over the

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

system variables. In our prototype implementation we restrict ourselves to mixed boolean and linear arithmetic. The EmbeddedPlatform provides a graphical editor together with a methodology to guide the engineer through the formalization process.

The SUP assumes discrete time with a fixed step size. An instance of the SUP observes the inputs and outputs of the system and emits a *failure* signal if the requirement is violated.

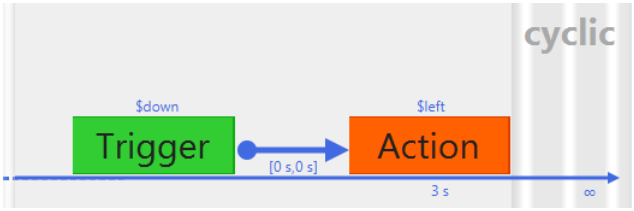

Fig. 1. SUP in the EmbeddedPlatform

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

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

The SUP consists of two parts, the *trigger* $(TSE, TC, TEE)[Tmin, Tmax]\,!\,TEC$ and the *action* $(ASE, AC, AEE)[Amin, Amax]!AEC$. Between trigger and action is the *local scope* $[Lmin, Lmax]$. The semantics of the SUP is controlled by the 15 parameters $TSE, TC, \ldots, AEC$ that are summarized in Table 1. The last parameter, maxTime, is not used in the examples in this paper and therefore not part of the arrow notation. Most of the parameters have a default value (third column in Table 1), e.g. per default $TSE = TEE = TC$ which allows to omit the TSE and TEE parameters in the notation (see Section 2.1). Most time parameters may be set to infinity (last column) to indicate an open time bound.

Besides the parameters, behavior of the SUP is controlled by its *interpretation*, *activation mode* and *startup phase*. The arrow notation introduced above is for the *progress* interpretation with startup phase *immediate* and activation mode *cyclic*. In the *progress* interpretation every occurrence of the trigger must be followed by the action. Because of the automata-based semantics however, after the first observation of the trigger following occurrences are ignored until the action is recognized. This is also called iterative observation [5]. In the interpretation *ordering*, every action must be preceded by a trigger. In the following we describe the *progress* interpretation. The interpretation *invariant* is a spe-

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

cial case of the *progress* interpretation where all parameters except TC, AC and maxTime are set to default values.

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. Hence, N is a parameter of type Time.

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

The *activation mode* controls if there are multiple observation cycles. There are three activation modes *init*, *first* and *cyclic* that are explained in the following together with the details of the SUP.

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.

Exit Conditions. If the *Trigger Exit Condition (TEC)* occurs before or at the TEE, the current observation cycle is aborted. In activation mode *cyclic*, a new observation cycle starts. The same occurs if the *Action Exit Condition (AEC)* occurs between (or at) TEE and AEE. The exit conditions have preference over success or failure of the SUP.

Global Scope. The SUP observation may be limited by the *global scope*. The global scope is an additional time parameter $maxTime$. $maxTime$ steps after the startup phase, the observation ends without result.

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.

Interpretation ordering. 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 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.

2.1 Examples

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

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

and $p[5] \rightarrow (q, r, s)[0, \inf)$ for

$$(p, p, p)[5, 5] \,!\, \mathit{false} \xrightarrow{[0,0]} (q, r, s)[0, \inf) \,!\, \mathit{false} \quad .$$

In the following we formalize the example 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.

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

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.

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

2.2 Formal Semantics

In the EmbeddedPlatform, formal requirements are translated into *observers*. Observers monitor executions of a system and indicate if a requirement is violated. In this paper, observers are modeled as finite automata with counters, called counter automata[2] [11], that operate on traces over the system variables.

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

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

Counter automata consist of finite sets of states, transitions and counter variables. Every transition is labeled with a boolean expression over observed system 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 [11]). *A counter automaton over a set $\mathbb{X}$ of variables is a tuple $\mathcal{A} = \langle S, \mathbb{X}, \mathbb{W}, I, F, T\rangle$ with states S, integer counter variables $\mathbb{W}$, initial and failure state $I, F \in S$ 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 $\gamma : \mathbb{W} \rightarrow \mathbb{N} \cup \{INC, STABLE\}$.*

A trace σ over $\mathbb{X} \cup \mathbb{W} \cup \{s\}$ with $\mathcal{V}_{type(s)} = S$ is a run for $\mathcal{A}$ if $s_0 = I$, $c_0 = 0$ for $c \in \mathbb{W}$ and for all $i \in \mathbb{N}$ exists $\langle s, g, \gamma, s'\rangle \in T$ such that $s_i = s$, $s_{i+1} = s'$,

[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].

$$s_i \models g \text{ and } c_{i+1} = \begin{cases} \gamma(c) & \text{if } \gamma(c) \in \mathbb{N} \\ c_i + 1 & \text{if } \gamma(c) = INC \\ c_i & \text{if } \gamma(c) = STABLE \end{cases} \quad \text{for } c \in \mathbb{W}.$$ *A run is accepting if* $s_i \neq F$ *for all* $i \in \mathbb{N}$.

We write $\sigma \models \mathcal{A}$ *if there is an accepting run* σ' *for* $\mathcal{A}$ *with* $\sigma' \downarrow_{\mathbb{X}} = \sigma \downarrow_{\mathbb{X}}$.

Here, $\sigma_i \models g$ indicates satisfaction of g on σ in step i. We assume that the set $Expr_{\mathbb{B}}(\mathbb{X} \cup \mathbb{W})$ allows at least boolean and linear integer arithmetic and comparison. 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.

The formal semantics [23] that have been kindly provided by BTC EmbeddedSystems to the author, use automata networks that are very close to counter automata. Compared to counter automata, these automata

- synchronize with each other by events
- may perform arbitrary many steps during one global (i.e. system-level) step, provided that each transition is taken at most once.

These automata networks can be translated to counter automata by constructing

1. the product of the single automata in the networkand
2. the transitive closure of the transitions possible during one global step.

Hence, we can use counter automata for the SUP semantics.

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

Technically, we construct an automata scheme for each combination of interpretation, startup phase, and activation mode of the SUP. An automata scheme is a counter automaton over the SUP parameters instead of the system variables. We derive $\mathcal{A}_O(R)$ from the automata scheme by substituting the parameters in the guards. Constructing the product and transitive closure of the automata network as sketched above, results in a blow-up of the automaton. Therefore we applied automata reduction techniques, such as unifying equivalent states, during construction of the automata schemes. These transformations are computationally expensive, but independent from the pattern parameters, so they need to be performed only once and the resulting automata schemes are reused for every instance of the SUP. In addition we apply some cheaper simplification rules to the derived counter automata.

Example 1. Figure 2 shows an observer automaton for **Req 1**.

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

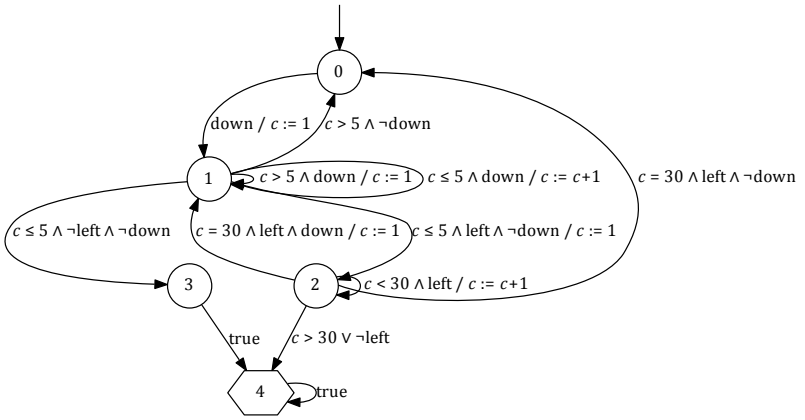

Fig. 2. Counter automaton for **Req 1**. State 0 is the initial and 4 the failure state.

the SUP is expressive enough to model industrial use cases. There is a case study at BOSCH [22] using a Specification Pattern System (SPS) originally proposed by Dwyer et. al. [10] and extended by Konrad and Cheng [19]. 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 $\sigma \in \mathcal{T}(\mathbb{X})$ 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: $\mathbf{F}_{\sim c}\phi :\Leftrightarrow$ 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. We use the reversed arrow $\leftarrow$ to indicate use of the *ordering* interpretation and keywords `first` and `initial` to denote deviant activation modes. The *last* operator, that is supported

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

Table 2. Expressing SPS patterns in SUP

both in the EmbeddedPlatform as well as our analysis prototype, refers to the value of P in the previous step. The MTL semantics of time-constrained SPS patterns is taken from [4], for untimed patterns the original LTL semantic [10] is used. 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 semantic in Table 2 is for the global scope, meaning without restriction by events.

We observe that the SUP does not have scopes ended by events, i.e. "between P and Q" and "until Q"[3] The global scope and "after P" directly correspond to startup phases of the SUP. Taking into account these missing scopes, we can formalize 90% of the SPS requirements from the BOSCH case study with the SUP. Compared to the iterative semantics of the SUP where multiple occurrences of the trigger before the action are ignored, in MTL every occurrence of the trigger is handled. However, for the patterns presented above the SUP observers precisely represent the discrete-time MTL semantics.

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 among others for completeness and determinism of the specification. In contrast to other approaches, the consistency checker for SCR performs only static checks, but no

[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.

reachebility analysis [15]. Similar methods are proposed for requirements state machines [17].

While the checker for SCR checks consistency locally for every state of the system, 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. Here, a set of requirements is called consistent, if there exists at least one trace that satisfies all requirements. 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 $\mathcal{M} \models R_1 \wedge \cdots \wedge R_n$. 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] consistency is based on the reachable states of the system. A state is called consistent if for every input exists an output of the system such that all requirements are satisfied and the next state is again consistent. The system is consistent if the initial state is. 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 [21] 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 [18]. 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 extends 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 2.1: 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. Inconsisten-

cies 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 $\mathcal{R}$ of SUPs over system variables $\mathbb{X}$, the set of satisfied traces by $\mathcal{T}(\mathcal{R}) = \{\sigma \in \mathcal{T}(\mathbb{X}) \mid \sigma \models \mathcal{A}_O(R) \text{ for } R \in \mathcal{R}\}$.

Definition 4 (strong consistency). *A set $\mathcal{R}$ of requirements is* strongly consistent *wrt. the system inputs $IN \subseteq \mathbb{X}$ if there exists $\Sigma \subseteq \mathcal{T}(\mathcal{R})$ such that*

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

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 *partial consistency* analysis focuses on those SUPs that we call *reactive*: An SUP is reactive if it has interpretation `invariant` or `progress` but is not of the form $\mathit{true} \xrightarrow{[0,0]} P$. All other requirements are called *invariant* in the following. We designed the partial consistency analysis with two ideas 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 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. For two SUPs R_1, R_2 the actions must occur at the same time if, for the time points t_1, t_2 marking completion of the trigger phases,

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

where $LS_R = [Lmin_R, Lmax_R]$ is the local scope and $AD_R = [Amin_R, Amax_R]$ is the action duration of an SUP. Eliminating the quantifiers leads to

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

Example 2. For requirements **Req 1** and **Req 2** (see Section 2.1) 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 this property 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 an observer automaton $\mathcal{A}_O(R)$, and some trigger automaton $\mathcal{A}_T(R)$. A trigger automaton has an accepting state instead of a failure state that is entered when R is triggered for the first time.

Definition 5 (trigger automaton). *A* trigger automaton $\mathcal{A}_T = (S, \mathbb{X}, \mathbb{W}, \delta, I, F)$ *is defined analogously to a counter automaton, except that F is an accepting state. Runs for trigger automata are analogously defined to runs of counter automata except that σ is an accepting run if $s_i = F$ at some time point i.*

Definition 6 (partial consistency). *For requirements R_1, R_2, R define*

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

Two reactive requirements R_1, R_2 and a set $\mathcal{R}_{inv}$ of invariant requirements are partially consistent *if*

$$\begin{aligned}
\forall k \in \mathbb{N} : (\exists \sigma \in \mathcal{T}(\mathcal{R}_{inv} \cup \{R_1\}) : \texttt{Cond}_{\{R_1,R_2\},k}(\sigma) \\
\wedge \exists \sigma \in \mathcal{T}(\mathcal{R}_{inv} \cup \{R_2\}) : \texttt{Cond}_{\{R_1,R_2\},k}(\sigma) \\
\Rightarrow \exists \sigma \in \mathcal{T}(\mathcal{R}_{inv} \cup \{R_1, R_2\}) : \texttt{Cond}_{\{R_1,R_2\},k}(\sigma)) \ . \quad (1)
\end{aligned}$$

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 $\texttt{Trigger}_{\{R_1,R_2\},k}(\sigma)$ returns true if R_1 and R_2 are triggered in σ before step k such that the `Interfere` condition is satisfied. Since we assume that the action exit condition is an external event, we are interested only in traces, where the action is not aborted. This is encoded in $\texttt{NoExit}_R(\sigma)$, which ensures that the AEC does not occur after triggering R. 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 and the consequence, the delay between triggers must lie in the interval calculated above, and we are only interested in cases where all invariant requirements are satisfied.

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 try to find finite prefixes satisfying a weaker resp. stronger acceptance criterion.

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

Bounded acceptance We restrict ourselves to finite traces satisfying the requirements, i.e. $\mathcal{T}_k := \{\sigma \in \mathcal{T}_k(\mathbb{X}) \mid \sigma \models \mathcal{A}_O(R) \text{ for } R \in \mathcal{R}\}$.

Searching for a loop [11] We introduce a modified acceptance relation $\models_{loop(i,j)}$ with $i < j \in \mathbb{N}$: For some finite trace σ, $\sigma \models_{loop(i,j)} \mathcal{A}$ if there is a finite accepting run σ' of length j for $\mathcal{A}$ such that $\sigma{\downarrow_{\mathbb{X}}} = \sigma'{\downarrow_{\mathbb{X}}}$ and $\sigma'_i = \sigma'_j$. We use this notions analogously for (sets of) SUP instances, i.e. $\mathcal{T}_{(i,j)} = \{\sigma \in \mathcal{T}_j(\mathbb{X}) \mid \sigma \models_{loop(i,j)} \mathcal{A}_O(R) \text{ for } R \in \mathcal{R}\}$.

Since the failure state of a counter automaton is a sink, acceptance clearly implies bounded acceptance. 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. Hence, every prefix with a loop can be extended to an infinite trace by infinitely repeating the loop. For details see [11].

Definition 7 (Bounded partial consistency). *Two reactive requirements R_1, R_2 and a set $\mathcal{R}_{inv}$ of invariant requirements are* (α, β)-bounded partially consistent *if*

$$\begin{aligned}\forall k \in \mathbb{N}, k \leq \alpha : (\exists i < j < k, \sigma \in \mathcal{T}_{(i,j)}(\mathcal{R}_{inv} \cup \{R_1\}) : \texttt{Cond}_{\{R_1,R_2\},i}(\sigma) \\ \wedge \exists i < j < k, \sigma \in \mathcal{T}_{(i,j)}(\mathcal{R}_{inv} \cup \{R_2\}) : \texttt{Cond}_{\{R_1,R_2\},i}(\sigma) \\ \Rightarrow \exists \sigma \in \mathcal{T}_{k+\beta}(\mathcal{R}_{inv} \cup \{R_1, R_2\}) : \texttt{Cond}_{\{R_1,R_2\},k}(\sigma)) \ .\end{aligned}$$

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

Corollary 1. *Partial consistency implies bounded partial consistency.*

Proof. Assume the left-hand side of the implication sign ($\Rightarrow$) in Definition 7 holds for some $k \in \mathbb{N}$. Because acceptance with a loop implies acceptance and, by definition, $\texttt{Cond}_{\{R_1,R_2\},i}(\sigma)$ implies $\texttt{Cond}_{\{R_1,R_2\},k}(\sigma)$ for $i < k$. Hence the left-hand side of the implication in eq. (1) also holds. Assuming partial consistency holds, also the right-hand side of eq. (1) holds with some trace σ. Since acceptance implies bounded acceptance, and $\alpha + \beta > k$, the right-hand side of Definition 7 holds with the prefix $\sigma{\downarrow_{\alpha+\beta}}$ as well.

Theorem 1. *If the trigger automata of R_1, R_2 depend only on input variables $X \in IN$, then strong consistency implies partial consistency.*

Proof. Assume the premise of partial consistency holds, i.e. for some $k \in \mathbb{N}$ there exists some trace τ such that

$$\tau \models \mathcal{R}_{inv} \cup \{R_1\} \wedge \texttt{Cond}_{\{R_1,R_2\},k}(\tau) \ .$$

By the definition of strong consistency, there is some $\sigma \models \mathcal{R}$ such that $\sigma{\downarrow_{IN}} = \tau{\downarrow_{IN}}$ (note that $\sigma'{\downarrow_0} = \tau{\downarrow_0}$ holds for arbitrary $\sigma' \in \Sigma$). Because the trigger automata for R_1, R_2 depend on IN only, $\texttt{Cond}_{\{R_1,R_2\},k}(\sigma)$ holds. As a consequence, the requirements are partially consistent.

Although partial consistency is based on triggered existential consistency, they are incomparable. The example from the introduction is existentially consistent but not partially consistent, as shown in the next section. On the other hand, the running example from [11] is partially consistent but not triggered existentially consistent. Note also that bounded triggered consistency implies triggered consistency while it is the other way round for partial consistency.

6 Implementation

We implemented a prototype of the partial consistency using the general-purpose SMT solver Z3 [20] 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 7.

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 $X_0, \ldots, X_n$ 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) and bound k is in $\mathcal{O}(k(|\mathcal{A}_T(R_1)| + |\mathcal{A}_T(R_2)| + \sum_{R \in \mathcal{R}_{inv} \cup \{R_1, R_2\}} |\mathcal{A}_O(R)|))$.Since by definition $\texttt{Cond}_{\{R_1,R_2\},k}(\sigma)$ implies $\texttt{Cond}_{\{R_1,R_2\},k+1}(\sigma)$, we have to check the consequence for the smallest k for that both parts of the premise are satisfiable only.

The encoding choosen in [11] introduces for every observer or trigger automaton $\mathcal{A}$ a boolean variable $\texttt{fail}_{\mathcal{A}}$ resp. $\texttt{fair}_{\mathcal{A}}$ that is *true* if the automaton is in its failure resp. accepting state in the current step. We found a trace satisfying some set $\mathcal{R}$ of requirements if $\texttt{fail}_{\mathcal{A}_O(R)} = \mathit{false}$ 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 $\texttt{fair}_{\mathcal{A}_T(R)}$ becomes true and remains stable in the rest of the trace. Then we encode $\texttt{Trigger}_{\{R_1,R_2\},k}$ in the target as

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

We enforce $\texttt{NoExit}_R$ by adding $\texttt{fair}_{\mathcal{A}_T(R)} \Rightarrow \neg AEC_R$ to the transition relation.

We encode satisfiability with a loop by introducing a copy s_{store} for each state variable s. Here the state variables for some automaton are 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. We reached a loop in the last step if the state variables again equal the copy variables.

Set	Premise 1	Premise 2	Consequence
Req 1, Req 2	19.46s	21.90s	7.86s
Req 1, Req 3	18.51s	0.37s	2.26s
Req 1, Req 4	18.73s	0.32s	5.66s
Req 2, Req 3	17.95s	0.31s	4.53s
Req 2, Req 4	16.85s	0.35s	2.25s
Req 3, Req 4	4.45s	–	–

Table 3. Solver run times for the analysis

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 $true \rightarrow \neg(up \wedge down)$

The analysis finds three inconsistent pairs in the example: $\{\textbf{Req 1}, \textbf{Req 2}\}$, $\{\textbf{Req 1}, \textbf{Req 4}\}$, and $\{\textbf{Req 2}, \textbf{Req 3}\}$. The run times for the Z3 (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 promise 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. 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 example from [6] in 80s. However, at time writing this paper we are not able to preserve the formal soundness results from Section 5 for symbolic time although we did not encounter any false findings of the analysis 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[5] STIMULUS tool [18] is another promising and interesting approach. In this example, a simulation engine is able to find the same inconsistencies as our approach. However, simulation-based techniques are less powerful in resolving non-determinism. As our notion of consistency is designed for high level requirements, these may have a high amount of non-determinism.

[5] https://argosim.com/

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 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.

References

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. 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)
18. 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)
19. Konrad, S., Cheng, B.H.: Real-time specification patterns. In: Proceedings of the 27th international conference on Software engineering. pp. 372–381. ACM (2005)
20. 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)
21. 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)
22. 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)
23. Teige, T.: Simplified Universal Pattern Syntax and Semantics. BTC EmbeddedSystems (06 2017), confidential
24. Zowghi, D., Gervasi, V.: On the interplay between consistency, completeness, and correctness in requirements evolution. Information & Software Technology 45(14), 993–1009 (2003)