Modeling Requirements for Quantitative Consistency Analysis and Automatic Test Case Generation*

Tom Bienmüller, Tino Teige, Andreas Eggers, and Matthias Stasch

BTC Embedded Systems AG, Gerhard-Stalling-Straße 19, 26135 Oldenburg, Germany
{bienmueller, teige, eggers, stasch}@btc-es.de,
http://www.btc-es.de

Abstract. We present improvements of software development processes based on formalized functional requirements. Fundamental basis is a graphical formalism called *simplified universal pattern* allowing users to model requirements using the intrinsic nature of functional requirements specifying a trigger/action relation. The underlying graphical formalism enables to generate additional benefits to existing processes. In this paper we focus on two techniques: quantitative requirement consistency analysis and automatic test case generation for functional requirements.

1 Introduction

Applying formal methods in software development processes requires *formalized temporal functional requirements* being available. Even though prominent formalisms such as LTL or CTL [5] exist, enabling non-formal-methods-experts to use them is still a major hurdle. Engineers need to be trained, stakeholders like quality managers demand to understand what has been expressed, and available formal requirements need to be revised after a new iteration cycle of the design has been initiated. Latest when it comes to real software *production*, non-functional requirements such as readability, understandability, maintainability become very important not only for the design being developed, but also for its formal functional requirement specification. Moreover, the *return on investment* for formal specifications may be doubted: *does it pay off when we spend time and money for establishing the needed skills and change our running processes?*

Making formal methods applicable in production therefore means, first, to bridge the gap between traditional requirements engineering and formalization, and, second, to add value to existing processes which clearly overcomes return on investment doubts.

In this paper, we address both of these key fundamentals: we describe the graphical *simplified universal pattern* formalism to *model* functional requirements in a natural, intuitive, and declarative way. Then we put a focus on two benefits: *automatic requirement consistency checking* and *automatic test case*

* This work has been supported by the ITEA3 project 14014 ASSUME.

Fig. 1. Example of a simplified universal pattern.

generation for functional requirements. We finally present initial experiments to prove the concept of our approach which is implemented in the commercial product suite BTC Embedded*Platform*®[1] for specification, testing, and verification of requirements for Simulink® and TargetLink® models as well as production code.

2 Modeling Functional Requirements

In the past, several graphical formalization languages have been proposed. *Symbolic timing diagrams* [10, 14] or *life sequence charts* (LSCs) [2] are prominent examples. Such formalisms offer tremendous expressiveness but this expressiveness comes along with needed expert knowledge to apply them. Approaches like those presented in [1] (*graphical pattern templates*) or [3] (*textual pattern templates*) limit expressiveness to gain better readability, but also benefit from reduced complexity for tableaux generation for formal verification. Though the latter two approaches use terms and notions to better understand the *formalism*, none of them inherently bases on the nature of an *informal functional requirement* which impose another hurdle to requirement engineers.

With over 20 years of experience in the field of formal specifications we believe it is mandatory to directly relate the formalism to the intrinsic composition of informal functional requirements. Furthermore, we believe it is important to not overwhelm end users with exorbitant expressiveness, which will also lead to better performance when reusing the artifacts later within formal techniques such as formal verification or consistency analysis. Many of our customers' functional requirements for system components can be described using a simple trigger/action relation, independent if they are expressing progress, ordering, or invariant requirements. The graphical language *simplified universal pattern* (SUP) proposed in this paper follows this path of building a formalism on top of the trigger/action relation. Note that approaches like LSCs or textual patterns could either *complement* or *enhance* the approach described here. LSCs are well suited to describe *interactions* between *integrated* components, while textual patterns are charming as these are easily accessible to humans. Chronologically, SUP is a further development from the graphical pattern templates [1] which have been used in previous BTC-ES' formal verification tools.

In the next subsection, we briefly recall the fundamental ingredients of that graphical formalism. More details can be found in [13] and [12].

[1] Product information can be found at `http://www.btc-es.de/` under "Products".

Name	Mode	Step 0	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9	Step 10
Formal Requirement	Status		Fulfilled							Violated	-	-
Commitment	SUP phase	Startup→▽(Tr)	→Tr→▽(Act)→Act→✓→▽(Tr)	▽(Tr)	→Tr→▽(Act)	▽(Act)	▽(Act)	▽(Act)	▽(Act)	→		
Sa1_driver_up	input	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Sa1_passenger_up	input	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Sa1_move_up	output	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Fig. 2. Tabular representation of an SUP run.

2.1 Simplified Universal Pattern

By long-standing experience and cooperation with engineers from prestigious automobile and aircraft manufacturers and suppliers, description languages for formalizing natural language requirements should be as *intuitive* as possible, *easy to understand*, and preferably presented in a *graphical* way such that formalization of human-readable to machine-readable requirements becomes a common engineering task without being very prone to errors. The *simplified universal pattern* (SUP) approach is based on the observation that the vast majority of real-life safety-critical requirements for components can be expressed by temporal trigger/action relationships like in the textual requirement "If the *driver up* or *passenger up switch is pressed* then the *window has to start moving up* in less than *50 ms*". An SUP explicitly introduces artifacts like *trigger* and *action* to close the gap between human intuition of a requirement and its formalized description, i.e. artifacts an requirements engineer talks about are directly reflected in the specification formalism, as shown in Fig. 1. We remark that a trigger or an action itself is not limited to be instantaneous but can have a temporal extent.

The semantics of an SUP is defined by *runs*, i.e. by (finite) executions of the system under test which are observed by an SUP. More precisely, a *trigger* or *action* is started by a run r at step i by consuming its *start event* from r at step i and *successfully passed* at step $j \geq i$ by accepting its *end event* at step j within the specified time interval, while its *condition* must hold in between, i.e. for all steps k with $i < k < j$. A *trigger* or *action fails* during processing if its condition became false or its end event was not observed in the time interval. An SUP is *fulfilled* by a run r if its trigger and action are successfully passed by r and their temporal relation is met. An SUP is *violated* by a run r if its trigger is successfully passed by r but the action does not start in the specified time interval or the action fails after entering it.

For a small example, consider the SUP from Fig. 1. One possible SUP run is shown in Fig. 2: in step 1 the expression of the trigger condition `Sa1_driver_up || Sa1_passenger_up` holds as `Sa1_driver_up` is true, and thus the trigger is passed. The SUP is then ready to observe the action which happens immediately as `Sa1_move_up` is also true in step 1. The SUP is fulfilled and waits for a new trigger. The next trigger is consumed in step 3 due to `Sa1_passenger_up`. Since the expression of the action condition `Sa1_move_up` does not hold in the following 5 steps/50 ms (where one step corresponds to 10 ms), the SUP is violated in step 8.

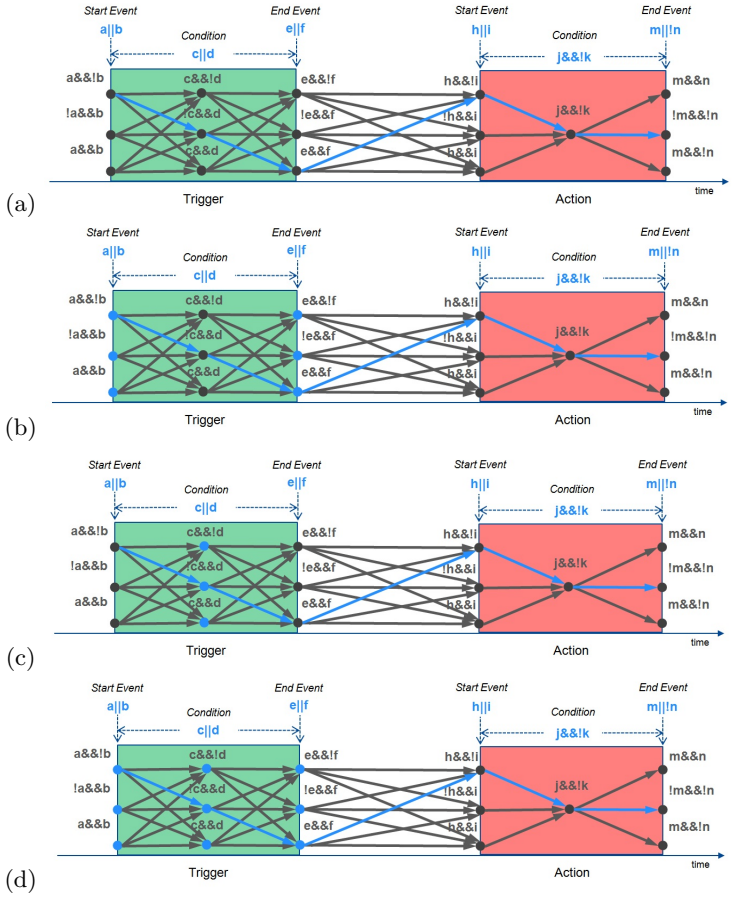

Fig. 3. Illustration of SUP coverage metrics: while Once (a) targets on finding one fulfilling run, TEC (b), TCC (c), and TE/CC (d) aim at covering their corresponding coverage goals being highlighted in blue.

2.2 Requirement Coverage

Explicitly revealing the intrinsic artifacts of an informal requirement through a formal SUP enables to define intuitive and accurately *measurable* coverage metrics for requirements. A commonly used informal coverage notion says that "there shall be a single test case linked to a requirement which verifies it". Whether the linked test case is actually doing that is not obvious. It requires a human review and confirmation. With coverage notions built on top of the SUP formalism, the above informal coverage metrics becomes clearly defined and measurable: executing the linked test case needs to generate a run which *completely traverses*, i.e. fulfills, the SUP once, in particular successfully passes the action end event.

Even though we are free to define arbitrary coverage notions based on SUP artifacts, we propose to use metrics first which only refer to the trigger part of a requirement, thus yielding the notion of *trigger coverage*. Coverage of the action part is left out but could be easily added. The proposed coverage metrics differ in the degree of exhaustion a trigger needs to be covered in order to reach the test exit criterion. Furthermore, we first restrict the coverage to fulfilling runs only, i.e. an SUP needs to be fulfilled by a run to induce coverage at all.

Once coverage (Once) intuitively corresponds to a metrics "one test for each requirement" and is achieved if there exists a run fulfilling the SUP.

The following more sophisticated coverage notions are defined based on a variant of *multiple condition coverage* (MCC). MCC is defined on all the atomic conditions occurring in an expression to be covered plus all their possible combinations. As the focus for requirement coverage is on fulfilling runs only, we restrict MCC to a subset we call *satisfying* MCC (sMCC), containing all combinations of conditions for which the overall expression evaluates to true.[2]

Trigger event coverage (TEC) is based on the sMCC coverage for the boolean expressions of the *trigger start* and *end events* of an SUP, while *trigger condition coverage* (TCC) is focussed on sMCC coverage goals of the *trigger condition*. *Trigger event/condition coverage* (TE/CC) combines both TEC and TCC.

For a coverage metric $\mathcal{C} \in \{\text{Once}, \text{TEC}, \text{TCC}, \text{TE/CC}\}$, an SUP S, and a set R of runs, we define the *coverage measure* $\mathcal{C}(S, R) \in [0, 1] \subset \mathbb{R}$ as follows. If $\mathcal{C} = \text{Once}$ then $\mathcal{C}(S, R) = 1$ if there is a run $r \in R$ that fulfills the SUP S, $\mathcal{C}(S, R) = 0$ otherwise. If $\mathcal{C} \neq \text{Once}$ then $\mathcal{C}(S, R) = c/g$ where c is the number of $\mathcal{C}$-goals for S covered by runs in R while g is the total number of $\mathcal{C}$-goals for S. An SUP S is called *fully $\mathcal{C}$ covered* by a set S of runs iff $\mathcal{C}(S, R) = 1$.

An illustration of above mentioned coverage metrics is given in Fig. 3. We remark that an analogous definition is conceivable to establish the notion of *action coverage* and moreover a combined *trigger/action coverage*.

3 Requirement Consistency and Test Case Generation

"Front loading" becomes more and more important. Hence, the quality of requirement specifications is obviously of tremendous relevance: the higher the quality of the requirements, the lower the probability of late iterations due to inconsistencies and incompletenesses of these requirements and their derivate artifacts. With formalized requirements using the SUP formalism and its contained requirement artifacts we can support front loading processes by applying dedicated requirement consistency analysis techniques on top of formalized requirements. Roughly, the set of runs induced by each SUP can be brought into relation to figure out, e.g., that requirements are contradicting. Then, the intersection of runs would be empty. Additionally, as we have detailed knowledge about each of the requirements, we even can give more *quantitative* information about a common implementation of those requirements. Though there might be

[2] In contrast to MCC, which induces 2^n coverage goals for an expression consisting of n atomic boolean conditions, sMCC induces $0 \leq i \leq 2^n$ goals.

runs that fulfill all requirements, *parts* of the requirements might be contradicting. This could be made visible based on the coverage notions from Sec. 2.2.

3.1 Quantitative Requirement Consistency

In the literature, topics like requirements consistency, completeness and correctness are quite rarely addressed in a formal sense. Survey papers like [4] show different informal but intuitive interpretations of these terms in different domains. In context of formal specifications the term *inconsistency* frequently refers to the fact that requirements are in conflict s.t. no valid system run exists, cf. [6]. The authors of [7] go some steps further and also take into account "whether timing bounds of real-time requirements may be in conflict" leading to the notion of *rt-inconsistency*. As requirements are often modeled by means of pre- and post-conditions as in SUP, the term of *vacuity* [8] even considers conflicting pre-conditions. In the following, we propose a new consistency notion called *basic SUP consistency* combining the ideas of *consistency* and *non-vacuity* but further incorporates a *quantitative* measure, namely by means of *requirement coverage* from Sec. 2.2.

Let be given a set $\mathcal{S} = \{S_1, \ldots, S_n\}$ of SUPs. We say that some SUP S_i is *basically SUP consistent for a run r wrt the remaining SUPs $\mathcal{S} \setminus \{S_i\}$* iff S_i is fulfilled by r and the remaining SUPs $\mathcal{S} \setminus \{S_i\}$ are not violated by r. We further call S_i *basically SUP consistent for a set R of runs wrt $\mathcal{S} \setminus \{S_i\}$, a coverage metric $\mathcal{C}$, and a coverage threshold θ* iff S_i is basically SUP consistent for each $r \in R$ wrt $\mathcal{S} \setminus \{S_i\}$ and the coverage measure $\mathcal{C}(S_i, R)$ according to metric $\mathcal{C}$ for the SUP S_i wrt the set R meets the coverage threshold θ, i.e. $\mathcal{C}(S_i, R) \geq \theta$. We finally define that a set $\mathcal{S}$ of SUPs is *basically SUP consistent wrt a coverage metric $\mathcal{C}$ and a coverage threshold θ* iff for each SUP $S \in \mathcal{S}$ there is a set R of runs s.t. S is *basically SUP consistent for R wrt $\mathcal{S} \setminus \{S\}$, $\mathcal{C}$, and θ*.

3.2 Automatic Test Case Generation

The SUP coverage definition straight forward leads to automatic generation of test cases for functional formal requirements. Generating a functional test from an SUP reduces to a proof task for a model checker claiming that an SUP can not be traversed completely while taking the coverage metric into account. If a corresponding counter witness exists, then this run fulfills the SUP and therefore can be viewed as a functional test for the corresponding requirement. We remark that within BTC Embedded*Platform*® we rely on model checkers based on SAT, SMT, and BDDs, cf. [11] and [9].

For this type of automatic test case creation, different interesting application scenarios exist. These scenarios differ in the definition of the "surrounding" of a formal requirement for which test cases shall be generated. Theoretically, we need to have a definition of runs available which describe the behavior of the system under test. An obvious source of runs is the system under test itself: a test generation takes only those runs into account which are induced by its (operational) implementation. The advantage is, that the generated test cases

will be functionally reasonable. The drawback is that test cases are generated from the system which shall be independently tested. Another option is to specify the "surrounding" by a set of formal requirements constraining the set of runs to a reasonable size – giving the advantage that the approach is independent of the system under test and can therefore be initiated in parallel to an implementation process. Here, the drawback is, that one is required to provide a "reasonable amount" of formal requirements in order to obtain the desired functional tests.

4 Example

The main goal of any consistency analysis is to find out early if requirements contradict with each other and hence to avoid that no controller can be built that satisfies them all. In the one extreme, one could look for a tool which checks if requirements contradict each other in every situation. This is probably the easiest analysis that can be performed, but also leads to the weakest consistency notion, since there could still be many situations in which more subtle contradictions exist. The other extreme is a scenario where there would be no contradiction for all situations (i.e. for every input or parameter combination in every temporal order). This analysis and methodology is the most costly, as it requires both the most computation complexity but also requirement refinement effort upon detected inconsistencies.

We therefore strive for an analysis laying between these two extremes. We propose to analyze formalized requirements encoded as SUP by means of basic SUP consistency from Sec. 3, which facilitates a quantitative aspect of requirements consistency. As a nice side effect, this approach generates functional test cases for requirements under consideration as mentioned in Sec. 3.2. For reasons of space, we use the following syntax for SUPs throughout this section:

$$\underbrace{\mathit{trigger}}_{\delta_{\text{trigger}}} \quad \underset{\Delta}{\longrightarrow} \quad \underbrace{\mathit{action}}_{\delta_{\text{action}}}$$

with *trigger* being the trigger condition, δ_{trigger} the optional trigger duration, *action* the action condition with its optional duration δ_{action}, and Δ the "local scope", i.e. the duration between trigger and action. A slightly extended form of the requirement from Fig. 1 is thus written as follows:

$$\text{driver\_up} \;||\; \text{passenger\_up} \quad \underset{[0,50]\,\text{ms}}{\longrightarrow} \quad \underbrace{\text{move\_up.}}_{50\,\text{ms}} \tag{S1}$$

We define a second requirement that shall enforce that the window moves down for 50 ms at most 10 ms after an obstacle is detected:

$$\text{detection\_obstacle} \quad \underset{[0,10]\,\text{ms}}{\longrightarrow} \quad \underbrace{\text{move\_down}}_{50\,\text{ms}} \tag{S2}$$

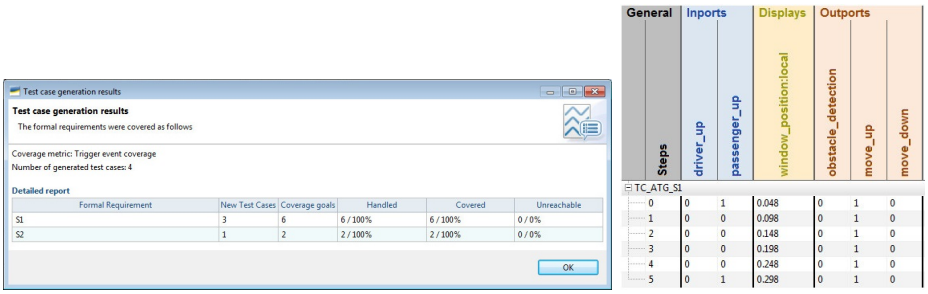

Fig. 4. Results of basic SUP consistency analysis and test case generation.

The goal of our tool-supported analysis is now to check basic SUP consistency of these two requirements to prove whether a correct implementation for them is possible or not. In order to model the environmental effects of the actuators move_up and move_down which might reveal a potential inconsistency of the requirements, we need to encode relevant behavior of the environment. The window moves up when the move_up output is set to true and the end-stop position is not yet reached:

$$\text{move\_up} \,\&\&\, (\text{window\_position} < 0.4) \quad \underset{[10,10]\,\text{ms}}{\longrightarrow} \quad \begin{array}{l}\text{window\_position} == \min(0.4,\\ \text{last}(\text{window\_position}) + 0.05)\end{array}$$

where $\text{last}(x)$ denotes the value of x at the last sample point. In our example, the sample time is set to 10 ms. It moves down when the move_down output is set to true while above the bottom:

$$\text{move\_down} \,\&\&\, (\text{window\_position} > 0) \quad \underset{[10,10]\,\text{ms}}{\longrightarrow} \quad \begin{array}{l}\text{window\_position} == \max(0,\\ \text{last}(\text{window\_position}) - 0.05)\end{array}$$

It does not move at the top, bottom, or when no actuator output is set:

$$\begin{array}{l}(!\text{move\_up} \,\&\&\, !\text{move\_down}) \,||\\ (\text{move\_down} \,\&\&\, (\text{window\_position} \leq 0)) \,||\\ (\text{move\_up} \,\&\&\, (\text{window\_position} \geq 0.4))\end{array} \quad \underset{[10,10]\,\text{ms}}{\longrightarrow} \quad \begin{array}{l}\text{window\_position} ==\\ \text{last}(\text{window\_position})\end{array}$$

4.1 Automatic Consistency Analysis and Test Case Generation

We have implemented an automatic check for basic SUP consistency in combination with automatic test case generation within BTC Embedded*Platform*®. In our example, we are particularly interested in basic SUP consistency of the SUPs S1 and S2. As the triggers are instantaneous, we chose as coverage metric $\mathcal{C} = \text{TEC}$.[3] The results of the automatic analysis are shown in Fig. 4: both SUPs S1 and S2 are *fully* basically SUP consistent, i.e. even threshold $\theta = 1.0$ is met, cf. table on the left. A generated test case for S1 is depicted on the right of Fig. 4: after having pressed passenger_up the move_up signal holds for 50 ms.

[3] Note that for instantaneous triggers, the expressions of the start and end events are equal. This implies that the derived coverage goals are the same for both events but are nevertheless reported separately.

4.2 Towards Controller Integration

The above example showed that basic SUP consistency defines a reasonable notion of requirement consistency. One obtains the information that a controller implementation exists for an analyzed set of requirements. Complementary, functional tests measurably covering requirements are automatically generated.

On the other hand, the example also shows limitations of basic SUP consistency. Even though the specified requirements are proved to be basically SUP consistent with 100% requirement coverage for the single requirements, there could be still inconsistencies in which could lead to problems when implementing a controller strategy for those requirements. In the above case, no controller exists which is able to handle some window-up request (driver_up or passenger_up) together with some obstacle detection (detection_obstacle). Please note that this type of inconsistency is an intuitive one and requires human validation – no automatism can judge whether it describes a relevant scenario, i.e. whether a controller implementation needs to deal with the mentioned inconsistent cases. It could be guaranteed by the integration of the controller that detected input inconsistencies can not occur. Here, computed inconsistencies would be irrelevant for a subsequent controller design and could be ignored.

It would be beneficial to have another notion of consistency available which is able to reveal potential inconsistencies based on specific input valuations as mentioned above. From a controller's perspective, we want to know whether constraints to its *integration* exist. To enable this, we reuse the notion of basic SUP consistency. The only addition is to derive a dedicated SUP which combines all triggers of the defined functional SUPs in order to express a context-free integration of the controller to be implemented. This additional SUP is called *integrity SUP* and is defined as follows:

$$\mathit{trigger}_1 \;||\; \ldots \;||\; \mathit{trigger}_n \qquad \longrightarrow \qquad \underbrace{1}_{\delta_{\text{action}}}$$

Applying test case generation for this integrity SUP requires to check all sMCC-combinations of all the SUP triggers of interest. If for some combination no run exists then this shows either demands on a controller's integration or an unwanted inconsistency has been detected. We need to remark that the current definition of an integrity SUP is only applicable if the triggers of the SUPs of interest are *instantaneous*. Moreover, the time duration δ_{action} need by given which currently is a manual task. In future work, we will thoroughly investigate the notion as well as the automatic derivation of *integrity* SUPs.

When considering the SUPs S1 and S2 of interest. Then, we can derive the integrity SUP S3:

$$(\text{passenger\_up} \,||\, \text{driver\_up}) \,||\, (\text{detection\_obstacle}) \qquad \longrightarrow \qquad \underbrace{1}_{100\,\text{ms}} \qquad \text{(S3)}$$

The result of the automatic consistency analysis for S3 is given in Fig. 5. It actually turned out that 6 of the 14 coverage goal are unreachable and thus

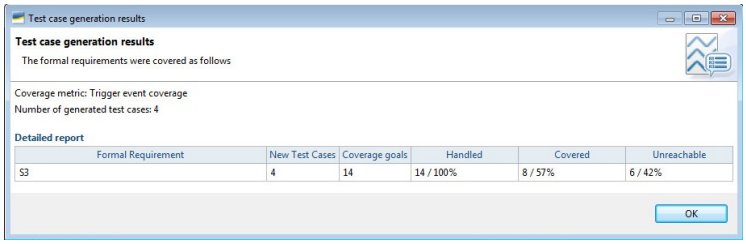

Fig. 5. Results of basic SUP consistency analysis for integrity SUP S3.

conflicts in behavior of the requirements are revealed. Due to the fact that the expressions of the trigger start and event event are equal (and thus their coverage goals), there are three conflicting situations remaining, namely exactly these with some window-up request together with some obstacle detection, more precisely (1) passenger_up && !driver_up && detection_obstacle, (2) !passenger_up && driver_up && detection_obstacle, and (3) passenger_up && driver_up && detection_obstacle. Based on this information, one can see that the simultaneous occurrence of passenger_up or driver_up with an obstacle detection causes situations in which no control strategy can satisfy both requirements. Based on this information, some kind of refinement can be performed, e.g. by relaxing the first requirement to enforce window movement only in case no obstacle is detected.

5 Conclusions and Future Work

In this paper we presented improvements of software development processes based on formalized functional requirements. Fundamental basis is a graphical formalism called simplified universal pattern (SUP), which enables users to model requirements using the intrinsic nature of functional requirements specifying a trigger/action relation. The underlying graphical formalism enables to generate additional benefit to existing processes. In this paper we focused on two methods, namely requirement consistency analysis and automatic test case generation for functional requirements, and proved the concept of these techniques by an example. In particular, when it comes to consistency analysis we motivated the necessity of introducing quantitative measures for being able to rate also controller integration demands.

In the current status of the technology we see several extension links in the future. Besides equipping consistency analyses with appropriate debugging facilities, we think about adding further quantitative measurements such as amount of runs fulfilling an SUP. We also need to consider *completeness* analyses of requirements, giving evidence whether "enough" requirements have been specified. By working closely together with industrial users we will collect the required feedback and needs and will let the tool evolve with respect to these needs.

Acknowledgments

The authors are very grateful to the anonymous reviewers for their helpful comments to improve the quality of the paper.

References

1. Bienmüller, T., Damm, W., Wittke, H.: The STATEMATE verification environment - making it real. In: CAV. LNCS, vol. 1855, pp. 561–567. Springer (2000)
2. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. FMSD 19(1), 45–80 (2001)
3. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE. pp. 411–420. ACM (1999)
4. Kamalrudin, M., Sidek, S.: A review on software requirements validation and consistency management. IJSEIA 9, 39–58 (2015)
5. van Leeuwen, J. (ed.): Handbook of Theoretical Computer Science (Vol. B): Formal Models and Semantics. MIT Press, Cambridge, MA, USA (1990)
6. Post, A., Hoenicke, J.: Formalization and analysis of real-time requirements: A feasibility study at BOSCH. In: Verified Software: Theories, Tools, Experiments. LNCS, vol. 7152, pp. 225–240. Springer (2012)
7. Post, A., Hoenicke, J., Podelski, A.: rt-inconsistency: A new property for real-time requirements. In: Fundamental Approaches to Software Engineering. LNCS, vol. 6603, pp. 34–49. Springer (2011)
8. Post, A., Hoenicke, J., Podelski, A.: Vacuous real-time requirements. In: International Requirements Engineering Conference. pp. 153–162. IEEE (2011)
9. Scheibler, K., Neubauer, F., Mahdi, A., Fränzle, M., Teige, T., Bienmüller, T., Fehrer, D., Becker, B.: Accurate ICP-based floating-point reasoning. In: Formal Methods in Computer-Aided Design (2016)
10. Schlör, R., Damm, W.: Specification and verification of system-level hardware designs using timing diagrams. In: European Conference on Design Automation. pp. 518–524. IEEE (1993)
11. Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., Bienmüller, T.: Successful use of incremental BMC in the automotive industry. In: Formal Methods for Industrial Critical Systems. LNCS, vol. 9128, pp. 62–77. Springer (2015)
12. Stasch, M.: Universal Pattern: Ein neuartiger Ansatz zur Visualisierung formaler Anforderungen. Master's thesis, University of Oldenburg (2014), in German
13. Teige, T., Bienmüller, T., Holberg, H.J.: Universal pattern: Formalization, testing, coverage, verification, and test case generation for safety-critical requirements. In: MBMV. pp. 6–9 (2016)
14. Wittke, H.: An environment for compositional specification verification of complex embedded systems. Ph.D. thesis, University of Oldenburg (2005)