Formalization and Analysis of Real-Time Requirements: a Feasibility Study at BOSCH

Amalinda Post[1] and Jochen Hoenicke[2]

[1] Robert Bosch GmbH, Stuttgart, Germany
amalinda.post@de.bosch.com
[2] University of Freiburg, Germany
hoenicke@informatik.uni-freiburg.de

Abstract. In this paper, we evaluate a tool chain to algorithmically analyze real-time requirements. According to this tool chain, one formalizes the requirements in a natural-language pattern system. The requirements can then be automatically compiled into formulas in a real-time logic. The formulas can be checked automatically for properties whose violation indicates an error in the requirements specification (the properties considered are: consistency, rt-consistency, vacuity). We report on a feasibility study in the context of several automotive projects at Bosch. The results of the study indicate that the effort for the formalization of real-time requirements is acceptable; the analysis algorithms are computationally feasible; the benefit (the detection of specification errors resp. the formal guarantee of their absence) seems significant.

1 Introduction

According to common industrial practice, requirements are specified in natural language and checked for errors manually, e. g., by peer reviews [16]. The shortcomings of this tool chain are well-known: the disambiguation of the (natural language) requirements is done by component specialists (instead of system specialists) during implementation and testing; both, the cost and the error detection rate of the manual checks do not scale well with the number of requirements, which each affect another and cannot be analyzed in isolation [3]. Further, a review can detect errors but never guarantee their absence.

A tool chain for the formalization of requirements and the (subsequently possible) formal, automatic analysis of requirements opens the perspective of eliminating the above shortcomings. Much research has been invested recently in language and tool support for both, formalization and analysis, e. g., [14,6,7,18,12,13]. The question whether such a tool chain is feasible in practice can not be decided by a principled argument that applies uniformly to all practical settings; we need a number of feasibility studies which address the question on a case-by-case basis. This paper presents such a study, for a special case of behavioral requirements, namely real-time requirements, in the

This is the author's version of the work published in *Verified Software: Theories, Tools, Experiments (VSTTE 12)*, LNCS 7125, pages 225–240. Springer, 2012. The orginal publication is available at
www.springerlink.com/index/10.1007/978-3-642-27705-4_18

context of several automotive projects at Bosch. We call a requirement *real-time requirement* if it contains an explicit timing bound, e. g., "If IRTest is set then the infrared lamps are turned on *after at most 10 s*."

We believe that real-time requirements are a good 'first target' for a feasibility study. Their formulation tends to be concrete; i. e., they are more amenable to formalization than other requirements. Real-time requirements are notoriously hard to get right, and they appear in projects for safety-critical systems; i. e., the extra need for quality assurance efforts is widely accepted. (The same reasons gave the incentive for previous work on real-time requirements [14,12,13]).

This paper contributes a first comprehensive evaluation of a tool chain for the algorithmic analysis of real-time requirements in a particular industrial setting. The tool chain contains, in addition to the algorithms proposed in [12,13], also a user-friendly input language which is indispensable in an industrial setting. In particular, we combine a specialized specification language based on a system of natural-language patterns [9,14] with analysis tools for requirements in a real-time logic [12,13]. To allow that we developed a compiler from the specification language into the real-time logic used in [12,13]; see Figure 1.

We perform a feasibility study in the context of several automotive projects at Bosch and evaluate the tool chain in terms of the human effort required for the formalization, the performance of the tools, and the outcome of the application of the tool chain to each of the examples in the case study. The overall result of the evaluation indicates that the tool chain is feasible and worthwhile. We will now recount the results of the evaluation in greater detail.

Results of the Evaluation. As expected, the effort for the formalization of real-time requirements was heavy. Two to three minutes per requirement in average seems still acceptable, however (in contrast with analysis), the formalization of requirements is done one by one; i. e., it scales linearly in the number of requirements. We chose the setting for our study where we start with already existing sets of documented informal requirements. This separation between requirements elicitation and formalization allows us to measure the effort spent for the formalization. The by far largest chunk of the measured effort goes into understanding the requirements. We thus obtain a safe approximation of the effort

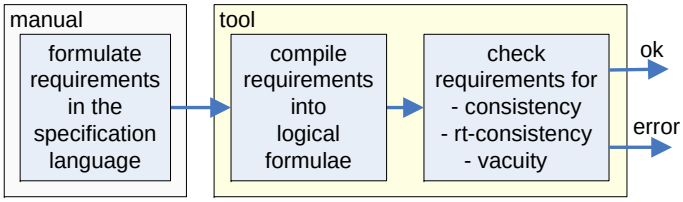

Fig. 1: The tool chain evaluated in the feasibility study. The compiler from the specification language to the real-time logic interfaces the manual formalization and the automatic analysis; see also Figure 2.

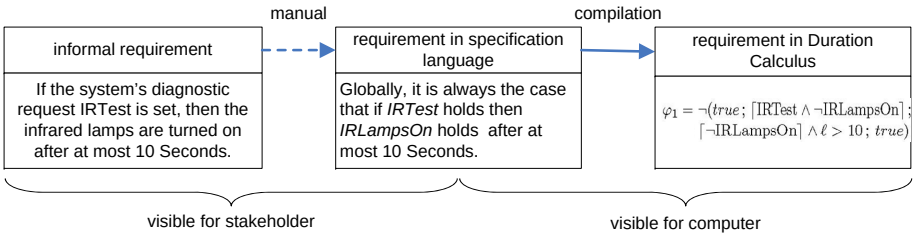

Fig. 2: In the tool chain, a specification language for real-time requirements is used as an intermediate step between the informal part and the formalism used as input for the analysis tools (the transition from the specification language to the input format is automatic, i. e., done by a compiler).

that is spent in the (preferable) setting where the formalization is interleaved with the elicitation.

We already knew that the computational complexity of the analysis algorithms is high and that the tools still need to be optimized [12,13]. The situation is comparable to model checking in that one cannot expect the tools to scale uniformly, and the tools need to be specialized to the application domain. The performance of the tools in our study (which ranges from several seconds to more than an hour) is too irregular for an on line use, e. g., interleaved with elicitation; for batch processing, as with our tool chain, the execution times on the examples in our study are more than acceptable.

The benefit of applying the tool chain seems significant. The detection of several specification errors in examples that had undergone extensive reviews is a benefit of obvious practical value. That is, the tool chain has allowed us to find errors that had escaped the reviews. Each error was detected because of the violation of one of three correctness properties. If we had a larger set of correctness properties that our tool chain could use, we might detect further errors still hidden in the requirements specification. This opens an avenue for further research.

The other kind of benefit, i. e., the formal guarantee of the absence of a particular kind of error, is of a purely conceptual value. It is interesting to note, however, that engineers at Bosch are quite keen on this functionality of the tool chain. This is noteworthy since engineers are reputed to be pragmatic. Perhaps mathematical certitude is an innate universal need, after all.

2 The Tool Chain

As depicted in Figure 1, the first step of the tool chain is manual and the second is fully automatic. In the first step, the requirements engineer formalizes real-time requirements in the specification language. The second step is a call to a tool (with, as front end, a compiler from the specification language to the proper input format of the analysis tools). We now briefly present the specification

language, the different properties checked by each of the analysis tools, and the analysis tools themselves.

In this paper, we rely on the results which state the correctness of the analysis tools which we use in our case study; for completeness, we will explain the properties checked by the analysis tools but we must refer to [14,12,13] for further details about the foundation of the algorithms used in the tools.

2.1 The Specification Language

The specification language is depicted in Table 1. It is a restricted English grammar based on the specification pattern system (SPS) given by Konrad and Cheng [9].

Every pattern consists of non-literal terminals P, Q, c and literal terminals. For example, in the *bnd response* pattern "it is always the case that if P holds, then S holds after at most c time unit(s)", P, S, and c are (the only) non-literal terminals. The non-literal terminals P and S denote boolean propositional formulae that capture properties of the system. The non-literal terminal c is instantiated with constants. In the example of a requirement (in the specification language) given in Figure 2, the pattern is instantiated by setting P to *IRTest*, S to *IRLampsOn*, and c to 10.

The specification language is geared toward a person who is not formally minded [14]. The use of the specification language in our tool chain is possible only thanks to the compilation from the specification language into the minimalistic formalism used for the input of the analysis tools; see Figure 2. The stakeholder only needs to care about the formulation of the requirements in the specification language; their formulation in the real-time logic (the input format of the analysis tools) is irrelevant for the stakeholder and only relevant for the analysis tool.

2.2 Translation of the SPS to Duration Calculus

Konrad and Cheng provide a translation of their SPS to the logics TCTL (Timed Computation Tree Logic), RTGIL (Real-Time Graphical Interval Logic) and MTL (Metric Temporal Logic). In this work we translate the SPS to the Duration Calculus fragment defined in [8]. Table 2 depicts our translation. These formulas are further translated into Phase Event Automata (PEA) [8], on which the consistency properties are checked.

Note that for the *eventually pattern* and the *response pattern* we need for the scopes *Globally, After Q, After Q until R* the translation uses the Duration Calculus operator $\triangleright$ introduced by Skakkebæk [15]. For these six instances the algorithms to check rt-consistency and vacuity cannot be directly applied, as the algorithm to calculate a PEA representing a requirement is not defined for requirements with the $\triangleright$ operator. We circumvent this problem in our tool in having defined the corresponding PEAs by hand.

Start	1: property	::= *scope specification* .
Scope	2: scope	::= Globally | Before R | After Q | Between Q and R | After Q until R
General	3: specification	::= *qualitative* | *real-time* | *invariant*
qualit.	4: qualitative	::= *absence* | *universality* | *existence* | *bnd existence* | *precedence* | *response*
	5: absence	::= it is never the case that P holds
	6: universality	::= it is always the case that P holds
	7: existence	::= P eventually holds
	8: bnd. exist.	::= transitions to states in which P holds occur at most twice
	9: precedence	::= it is always the case that if P holds, then S previously held
	10: response	::= it is always the case that if P holds then S eventually holds
real-time	11: real-time	::= *min duration* | *max duration* | *bnd recurrence* | *bnd response* | *bnd invariance*
	12: min dur.	::= it is always the case that once P becomes satisfied, it holds for at least c time unit(s)
	13: max dur.	::= it is always the case that once P becomes satisfied, it holds for at most c time unit(s)
	14: bnd recur.	::= it is always the case that P holds at least every c time unit(s)
	15: bnd resp.	::= it is always the case that if P holds, then S holds after at most c time unit(s)
	16: bnd inv.	::= it is always the case that if P holds, then S holds for at least c time unit(s)
invariant	17: invariant	::= it is always the case that if P holds, then S holds as well

Table 1: Restricted English grammar based on the grammar given by Konrad and Cheng in [9].

2.3 The Correctness Properties

In the tool chain we check requirements for three properties: *inconsistency, rt-inconsistency* and *vacuity*.

We say that a set of requirements φ is *inconsistent* if there exists no system satisfying φ, e. g., it exists no system satisfying both "Req_1: Once *IRTest* holds it holds for at least 5 Seconds." and "Req_2: Once *IRTest* holds it holds for at most 3 Seconds."

The check for *rt-inconsistency* analyzes whether timing bounds of real-time requirements may be in conflict. The formal definition of rt-inconsistency is given in [12], e. g., the following two requirements are *consistent* but not *rt-consistent*: "Req_3: Globally, it is always the case that if *IRTest* holds, then *IRLamps* holds after at most 10 seconds", "Req_4: Globally, it is always the case that if *IRTest* holds, then $\neg$*IRLamps* holds for at least 6 seconds". Say the observable *IRTest* holds from time point 4 on for 6 seconds (as depicted in Figure 3). Then Req_3

Table 2: Translation of the SPS into Duration Calculus (Excerpt)

Scope	Pattern	Duration Calculus
Globally	it is never the case that P holds	$\neg(true; \lceil P \rceil;\ true)$
Before R		$\neg(\lceil \neg R \rceil; \lceil \neg R \wedge P \rceil; \lceil \neg R \rceil;\ true)$
After Q		$\neg(true; \lceil Q \rceil; true; \lceil P \rceil;\ true)$
Between Q and R		$\neg(true; \lceil Q \wedge \neg R \rceil; \lceil \neg R \rceil; \lceil P \wedge \neg R \rceil; \lceil \neg R \rceil; \lceil R \rceil;\ true)$
After Q until R		$\neg(true; \lceil Q \wedge \neg R \rceil; \lceil \neg R \rceil; \lceil P \wedge \neg R \rceil;\ true)$
Globally	P eventually holds	$(\neg(\lceil \neg P \rceil)) \rhd true$
Before R		$\neg(\lceil \neg R \wedge \neg P \rceil; \lceil R \rceil;\ true)$
After Q		$(\neg(true; \lceil Q \wedge \neg P \rceil; \lceil \neg P \rceil)) \rhd true$
Between Q and R		$\neg(true; \lceil Q \wedge \neg R \rceil; \lceil \neg P \wedge \neg R \rceil; \lceil R \rceil;\ true)$
After Q until R		$\neg(true; \lceil Q \wedge \neg R \rceil; \lceil \neg P \wedge \neg R \rceil; \lceil R \rceil;\ true) \wedge (\neg(true; \lceil Q \wedge \neg P \wedge \neg R \rceil; \lceil \neg P \wedge \neg R \rceil)) \rhd true$
Globally	it is always the case that if P holds then S eventually holds	$(\neg(true; \lceil P \wedge \neg S \rceil; \lceil \neg S \rceil)) \rhd true$
Before R		$\neg(\lceil \neg R \rceil; \lceil P \wedge \neg S \wedge \neg R \rceil; \lceil \neg S \wedge \neg R \rceil; \lceil R \rceil;\ true)$
After Q		$(\neg(true; \lceil Q \rceil; true; \lceil P \wedge \neg S \rceil; \lceil \neg S \rceil)) \rhd true$
Between Q and R		$\neg(true; \lceil Q \wedge \neg R \rceil; \lceil \neg R \rceil; \lceil P \wedge \neg R \wedge \neg S \rceil; \lceil \neg R \wedge \neg S \rceil; \lceil R \rceil;\ true)$
After Q until R		$(\neg(true; \lceil Q \wedge \neg R \rceil; \lceil \neg R \rceil; \lceil P \wedge \neg S \wedge \neg R \rceil; \lceil \neg S \wedge \neg R \rceil)) \rhd true \wedge (\neg(true; \lceil Q \rceil; true; \lceil P \wedge \neg S \rceil; \lceil \neg S \rceil)) \rhd true$
Globally	it is always the case that if P holds then S holds after at most c time units	$\neg(true; \lceil P \wedge \neg S \rceil; \lceil \neg S \rceil \wedge \ell > c;\ true)$
Before R		$\neg(\lceil \neg R \rceil; \lceil \neg R \wedge P \wedge \neg S \rceil; \lceil \neg R \wedge \neg S \rceil \wedge \ell > c;\ true)$
After Q		$\neg(true; \lceil Q \rceil; true; \lceil P \wedge \neg S \rceil; \lceil \neg S \rceil \wedge \ell > c;\ true)$
Between Q and R		$\neg(true; \lceil Q \wedge \neg R \rceil; \lceil \neg R \rceil; \lceil P \wedge \neg R \wedge \neg S \rceil; \lceil \neg S \wedge \neg R \rceil \wedge \ell > c; \lceil \neg R \rceil; \lceil R \rceil;\ true)$
After Q until R		$\neg(true; \lceil Q \wedge \neg R \rceil; \lceil \neg R \rceil; \lceil P \wedge \neg R \wedge \neg S \rceil; \lceil \neg S \wedge \neg R \rceil \wedge \ell > c;\ true)$

requires that *IRLamps* appears not later than $t = 14$. At the same time Req_4 requires that *IRLamps* does not hold until at least $t = 16$—a conflict. Formally, a set of requirements is *rt-inconsistent* if there is a a finite trace satisfying all requirements that cannot be extended to an infinite trace.

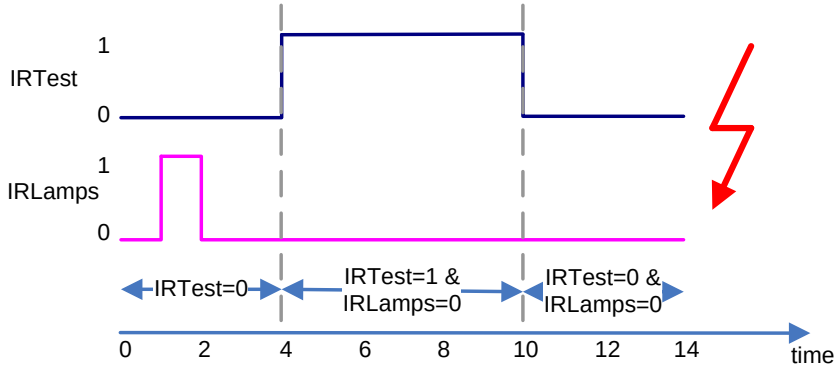

Fig. 3: Req_3 and Req_4 are rt-inconsistent.

The check for *vacuity* checks whether there is a requirement in the set that is only vacuously satisfied in the context of the set. The formal definition is given in [13], e. g., the following requirements are *consistent* and *rt-consistent* but *vacuous*: "Req_5 : Globally, it is always the case that if *IRTest* holds then *IRLamps* holds after at most 10 seconds", "Req_6: Globally, it is never the case that *IRTest* holds". In every system satisfying both requirements the observable *IRTest* never holds (according to Req_6), i. e., the precondition of Req_5 never holds. Thus, in the context of the set of requirements Req_5 is only *vacuously satisfied*. We assume that a requirements engineer specifies only requirements with behavior that shall be visible in a system, thus we assume that there is an error in the requirements and call Req_5 *vacuous* with the set of requirements.

To formally define vacuity, a purely syntactical characterization of *simpler* requirements is needed. In our case a requirement is of the form $\neg(\varphi_1; \ldots; \varphi_n; true)$ and simpler requirements are those, where some trailing phases $\varphi_i; \ldots; \varphi_n$ are omitted. Then a requirement φ is *vacuous* in the context of a set of requirement, if there is a simpler requirement that is equivalent in the context. For example, if in some context, the requirement "after Q it is never the case that P holds", $\neg(true; \lceil Q \rceil; true \lceil P \rceil; true)$, is equivalent to "it is never the case that Q holds", $\neg(true; \lceil Q \rceil; true)$, we say that the first requirement is vacuous in that context.

2.4 The Tool

We have assembled a prototype tool that parses requirements formalized in the specification language and automatically transforms them into formulae in a real-time logic (the Duration Calculus); it then analyzes the formulae to check the

formalized requirements for *consistency, rt-consistency* and *vacuity*; see Figure 4. The tool is written in Java and bases on the PEA-toolkit developed by [11] and the model checker UPPAAL [2]. Every algorithm is sound and complete (i. e., for every input data the decision procedure returns a correct answer). If the set of requirements is rt-inconsistent, the tool returns a counterexample (a possible behavior that leads to a timing conflict). If the set of requirements is incongruous then the tool returns the requirement that is incongruous in the set.

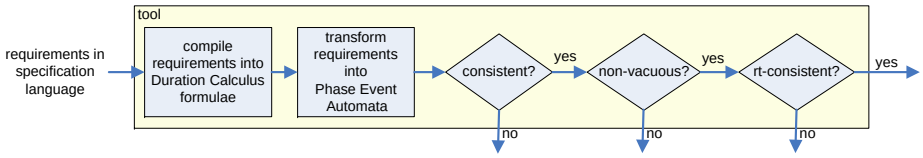

Fig. 4: The prototype tool compiles requirements in specification language to Duration Calculus formulae; it then starts the analysis tools, i. e., it transforms the logical formulae into a Phase Event Automaton and then checks for consistency, vacuity, and rt-consistency.

3 Planning of the Feasibility study

3.1 Study Goals and Questions

In order to assess the practical relevance of the tool chain in the automotive context, two main questions must be addressed. First, what is the benefit of the tool chain? Second, what are the costs of applying the tool chain?

A preliminary question is whether requirements engineers and software developers are in principle willing to use the specification language for requirements. We addressed this question in an informal inquiry, before starting the actual feasibility study which addresses the two questions above. We asked requirements engineers at Bosch to take some of their behavioral requirements and reformulate them in the specification language. We showed requirements formalized in the specification language to software developers at Bosch and asked them to explain their meaning to us. The reaction was only positive; i. e., the specification language seems easy to use, both for writing and reading. In the context of this inquiry, the request for tool support was constantly repeated.

Before developing a tool that is fit for an industrial use, we decided to first develop a prototype tool and start with evaluating the two questions above in a feasibility study on requirements of different Bosch projects. To summarize, we identified the following two items for evaluation in the study.

(Benefit) Is the tool chain useful in terms of quality assurance for requirements, resp., does it help to identify errors that were not detected in a manual review? Does it support a user in resolving the errors?

(Cost) What effort (measured in time) is needed to formulate requirements in the specification language? Are the analysis algorithms computationally feasible for the examples in the study?

3.2 Selection of the Sample

In the first step we selected requirements documents from different BOSCH projects of the automotive domain. To get a representative sampling, we decided to apply stratified sampling over the automotive application domains *driving assistance, engine controlling, car multimedia, catalytic converter development* and *power train* development. We then used convenience sampling to select a project out of every stratum.

Each project had several requirements documents, some consisting of more than 100 pages. In order to get a representative sample of requirements we asked the corresponding requirements engineers to give us 1 to 4 sets of requirements (representative for their domain and containing behavioral requirements and real-time requirements), each specifying a system component. This way we obtained sixteen sets of requirements.

3.3 Feasibility Study Design

In the first step we formulated the requirements in the specification language. Every set of requirements was then again reviewed with feedback from the responsible requirement engineers, and, if needed, changed until we agreed that the meaning of the informal requirements was accurately represented in the requirements formulated in the specification language. For one project we let the requirements engineer directly translate the requirements in SPS, in this case we were just available as coach. We then used the tool to automatically transform the requirements into Duration Calculus formulae. We checked the requirements with the help of our tool for *consistency, rt-consistency* and *vacuity* on a PC Windows XP system with 2 GHz Intel Core 2 Duo processor and 1 GB RAM, whereas only one core was used. If the tool detected an error we searched the reason for the error, fixed it and then checked the set again (until the set was consistent, rt-consistent and non-vacuous). We measured the execution time as CPU-time needed to parse the requirements, transform them to Duration Calculus formulae and then do the respective check.

4 Analysis of the Results

4.1 Benefit

Benefit Table 3 depicts the validation results for each component. The specifics of the components are not relevant; hence we do not present them and just number the examples from 1 to 16 (first column). The second column refers to the size of the input in the number of requirements. Columns 3 to 6 refer to the outcome of the consistency/vacuity/rt-consistency check.

	$\#_{req}$	consistent?	non-vacuous?	rt-consistent?
1	9	yes	yes	no
2	10	yes	n/a	no
3	10	yes	yes	no
4	12	yes	yes	yes
5	13	yes	yes	yes
6	17	yes	yes	yes
7	17	yes	yes	no
8	18	yes	yes	no
9	27	yes	yes	yes
10	27	yes	yes	yes
11	29	yes	yes	no
12	40	yes	yes	no
13	48	n/a	n/a	n/a
14	58	n/a	n/a	n/a
15	81	yes	no	yes
16	81	yes	yes	yes

Table 3: Checking consistency, rt-consistency and vacuity for existing examples of sets of real-time requirements for software components in automotive projects at Bosch using a prototype implementation (Fig. 4).

As Table 3 shows, every component (except Component 13 and 14) was consistent. We guess that this indicates that the manual review process successfully detected any inconsistencies. For Component 13 and 14 we got an out-of-memory error thus we could not determine the consistency.

Regarding vacuity, the automatic validation could guarantee the absence of vacuities for 12 components, in one component it detected an error. In Component 8 the precondition of the following requirement was never satisfied *"If* $accelerationPedal = 0$ *and brakePedalActivated then regeneration holds after at most 1 time unit."* Debugging the requirements we found out that another requirement was ambiguously specified as *"The value range of the acceleration pedal is between 0 and 100."* This second requirements was misinterpreted as "$0 < accelerationPedal < 100$" instead of "$0 \leq accelerationPedal \leq 100$". We resolved the ambiguity and changed the requirement to *"The value range of the acceleration pedal is between 0 and 100 where the endpoints of the interval are included"*. The needed change was only a minor change, but the analysis helped to discover an ambiguous requirement. Thus, the check for vacuity was beneficial. Note that only the biggest component contained an vacuity. We see two possible reasons for that. First, it might be that in practice vacuity only rarely occurs. Second, it might be that vacuities occurred, but they were already detected in the manual reviews and subsequently resolved. In the belief of the requirements engineers at Bosch, vacuity occurs in practice. Thus, we believe that the vacuities were resolved in the earlier steps. This would also explain, why the vacuity was detected in the biggest component—large components are

much more difficult to review for humans, as it gets difficult to keep the interdependencies in mind.

For three components our algorithm returned with an out-of-memory error. Note that the space of solution tends to explode, if there are not many interdependencies between the requirements in the set. If there are many interdependencies, then the space of solution gets reduced. Thus, the instances that are difficult to check for the tool, are often quite easy to review for a human—and vice versa: the instances with many interdependencies, which are more difficult to grasp for a human, get easy to check for the tool. Thus, for this property it seems that the automatic and the manual analysis might well complement each other.

Most errors were discovered by the rt-consistency-check: seven out of 16 components were in fact rt-inconsistent, i. e., for Components 1, 2, 3, 7, 8, 11 and 12, the check identified flaws in the requirement specification that needed to be repaired. Major changes were needed to correct these requirements. e. g., for Component 3, two of the existing requirements were deleted, five were changed, and seven new requirements were added.

Fixing the errors was an iterative process, with up to 10 iterations. In every iteration we thought to have fixed the problem, but then the check again found an rt-inconsistency. The output-interpretation then helped us to identify the reason of the errors. Thus, although rt-inconsistencies seem to appear frequently in requirements specifications this property seems to be difficult to detect for humans. The benefit of this check is thus very high.

For Component 4, 5, 6, 9, 10 and 15 the tool chain assured the consistency, non-vacuity, and rt-consistency of the requirements. The tool chain helped us to assure the quality of six components, and to detect 8 errors in the requirements that were not known before. Three of the errors were even found in the smallest sets of requirements. We thus think that the tool chain is even beneficial for smaller sets of requirements.

Costs To evaluate whether the benefit of applying our tool chain justifies its costs, we measure the time needed to formalize the requirements and second the execution time needed by our tool (i. e., the time the requirements engineer needs to wait for the results).

To express the informal requirements in the SPS we needed in average about 2–3 minutes per requirement, i. e., for Component 1 to 9 we needed 26 Min, 30 Min, 28 Min, 35 Min, 25 Min, 28 Min, 32 Min, 38 Min, 60 Min, and 2 h 50 Min for Component 16. Most of the time was needed to understand the meaning of the informal requirement, the reformulation itself was then quickly done. However even two minutes per requirement may scale to a considerable amount of time as in the automotive domain there are often thousands of requirements for one product. However, if the tool chain is integrated within the development processes (i.e, the requirements are directly formulated in the SPS when developing the requirements) then these costs could be omitted.

Secondly, we evaluate the execution time of the tool. If the requirements engineer needs to wait a long time before getting the validation results this is costly. As he is working on other topics in the meantime, he will need some time to familiarize himself again with the requirements, and it will need more time to debug the requirements. In practice, the requirements engineer needs to wait for the results of a manual review for some days. Thus, our tool chain has to compete with that time slot. The execution time for the checks is considerably smaller. The longest execution time took 1 h 32 Min—a big improvement. Thus, with respect to the waiting time, applying the tool chain might even decrease the costs of requirements engineering, as the requirements engineer can validate a set of requirements directly when specifying the requirements.

Over all, it seems that the tool chain is very beneficial and the costs of applying the tool chain are reasonable, they might even slightly decrease. However, there is one restriction: we suspect that for big sets of requirements the space of solutions grows explosively. For Component 13 and 14, our checks returned with an Out-of-Memory-Error. Further optimizations are needed to develop efficient algorithms, still the algorithms have to handle the state explosion problem [19,12]. We think that the tool chain is cost-effective when validating component requirements or sets of component requirements, but probably not suited to validate the set of all system requirements at once. Further studies on bigger sets of requirements are needed to confirm or refute that belief.

4.2 Observations

Usability. In an initial survey we had asked requirements engineers of Bosch to apply the SPS on requirements. The results indicated that they thought the SPS easy to apply and easy to learn. This was confirmed in our feasibility study: initially, the training curve was steep. For the first requirements the requirements engineer needed up to 10 minutes per requirement to express the requirement in SPS, i. e., to compare the requirement with the available patterns. But once he had used a pattern at least once the needed time considerably decreased to 1–3 minutes. Further, all requirements engineers could directly explain the meaning of a requirement in SPS. Only the use of the different scopes needed some explanation. Thus, we think that the SPS as input language is suited for the whole development team, even without much training.

In contrast, we believe that to interpret the output of the checks a more extensive training is needed. For a given set of requirements we return the check result, and if the set is rt-inconsistent a run to the rt-inconsistency, and if it is vacuous the set of requirements that are only vacuously satisfied. Without training, the engineers could interpret the check results, but they had some problems in interpreting the runs. But without the runs, the requirements are very difficult to debug. Thus, we think that the tool chain allows that many developers specify requirements, and they can do so directly in SPS. But we recommend that only the requirements engineer checks the requirements for consistency, rt-consistency and non-vacuity. This way, only the requirements engineer needs to be trained in interpreting the output of the checks.

For some requirements it was difficult to decide, whether the requirement should be formalized with an "invariant pattern" or with a "(bounded) response pattern", e. g., the requirement *"If the system is in error-mode then AssistFunction has to be deactivated"* might be formalized as *"Globally, it is always the case that if errorMode holds then ¬AssistFunctionActive holds as well"*, expressing the desired invariant relation between the two variables. However, on a deeper abstraction level, that may not be quite true. Say the system state is calculated in one software function, and *AssistFunction* is implemented in another function. Both functions are called in the same task, and *AssistFunction* checks the system state when being called. Then the formalization *"Globally, if errorMode holds, then ¬AssistFunction holds after at most 10 ms."* would be more appropriate. The natural language requirement may be the same on every abstraction level, although its meaning changes depending on the context. In contrast the formalized requirements must change depending on the abstraction levels—they make the change of the semantics explicit.

Specification Language. We further noticed that if a requirements engineer specified a system component in SPS, then the requirements in SPS tended to contain more information than the ones in natural language, e. g., the requirement *"If the locally measured voltage is not available for Local Voltage Usage (InternVoltageError), the system voltage value as received from the bus shall be used."* was expressed in SPS as *"Globally, it is always the case that if ¬BusOff ∧ InternVoltageError holds then VoltageValue'=VoltageValueFromBus eventually holds"*. The requirements engineer used the implicit knowledge, that the voltage value received from the bus is only received if the bus is not off. Thus, it seems that the SPS is a way to make implicit knowledge more explicit.

Nearly all requirements in the case study specified invariant or future behavior. The precedence pattern was only used once.Further, about two thirds of the requirements were formalized using the invariant pattern(*it is always the case that if P holds then S holds as well*) or (bounded) response pattern (*it is always the case that if P holds then S eventually holds,* resp., *then S holds after at most C time units*. The other patterns were only rarely used. We believe that this is the case, as for the specification of the behavioral requirements the systems were seen as blackbox. Thus, the resulting requirements mostly relate input and output variables to each other.

What we found a bit surprising was that there was no need to express uncertainty or prioritization in the specification language. When asking the requirements engineers, they said that all requirements needed to be implemented, there are no "nice-to-have" requirements. The only prioritization needed is the one to decide what requirement has to be implemented for what release—but this question was managed with the help of requirements attributes. Furthermore, although there is a high degree of uncertainty in the requirements, this uncertainty is hidden within application parameters, e. g., say *AssistFunction* shall only be active if the vehicle speed is above a certain threshold, however the value of that threshold is not yet known. Then, the requirements engineers invent an application parameter and specify the requirement like, e. g., *"if velocity ≤ threshold*

holds then ¬AssistFunctionActive holds as well". This way, uncertainty is resolved using application parameters.

Change Requests. We detected three items to tailor the specification language by Konrad and Cheng to the automotive domain. First, the requirements engineers asked for a new pattern, specifying invariant behavior. They wanted to express requirements like *if P holds then S holds as well*. In the specification language given by [9] this behavior can be expressed using the *absencepattern*, i. e., via *"it is never the case that P∧¬S holds"*. However, the requirements engineers thought it to be not intuitive, to specify invariant behavior via the absence pattern. We thus extended the specification language with the further pattern, to improve the intuitiveness. Second, some requirements engineers asked us to omit the "it is always the case that" in the *precedence, response, min duration, max duration, bnd recurrence-, bnd response* and *bnd invariance* pattern. They noted that this part of the pattern sounded strange in combination with the *Globally* scope. Third, we needed one further pattern. In the error handler concept, errors need to be qualified (i. e., they need to be detected during some time) before they are stored in the error memory. To express such a behavior we needed one further pattern, *"if P holds for at least c time units then S holds after at most c time units."*.

5 Threats to Validity

In this section, we analyze threats to validity defined in Wohlin [17]. Note that the results of the study are only valid in the given context, we do not aim to make any generalizations. Threats to validity concerning the suitability of the specification language are discussed in [14].

5.1 Construct Validity

Expectancy Effect. Expectations of an evaluator toward the outcome can affect a study. We formulated the informal requirements in the specification language. However, the reliability analysis in [14] suggests that applying the specification language is sufficiently independent of the evaluator. Furthermore, the requirements were automatically analyzed by a tool, thus the number of errors is objectively measured.

Inadequate Preoperational Explication of Constructs. This threat arises if the measures are not well defined. In order to minimize that threat we discussed with experts whether our properties *inconsistency, rt-inconsistency* and *vacuity* represent their concept of erroneous behavioral requirements. The discussion indicated that the properties seem to capture erroneous requirements. Furthermore, we formally (i. e., unambiguously) defined the properties. We obtained a safe approximation of the effort that is spent for the formalization and the check in separating elicitation, formalization, and analysis, i. e., in measuring the effort to formulate an informal requirement in specification language (independent of the elicitation effort) and the execution time of the analysis tools.

5.2 External Validity

Sampling validity. This threat arises if the sample is not representative for the requirements. The defects in the requirements specifications might over or under represent the defects in the rest of the corporate requirements specifications. In order to minimize this threat we used the selection procedure described in Section 3.2. A limitation of the feasibility study is that we only used requirements of BOSCH projects. Thus we cannot extend our results to the whole automotive domain.

6 Conclusion

The contribution of this paper is twofold. First, we showed how to build-up a tool chain that combines a user-friendly input language (based on a SPS) with an automatic check for consistency, rt-consistency and non-vacuity as proposed in [12,13] for requirements specified in the real-time logic Duration Calculus. Second, we have evaluated the tool chain to algorithmically analyze real-time requirements, in the context of several automotive projects at BOSCH. In particular, compared to the case studies in [12,13] we extended the number of samples to 16 samples from six samples in [12] and eleven samples in [13]. The results of the study indicate that the effort for the formalization of real-time requirements is acceptable, that the analysis algorithms are computationally feasible, and that the benefit (the detection of specification errors resp. the formal guarantee of their absence) seems significant. This is encouraging. However, before we can turn the tool chain into a technology that is apt for industrial use, we need to solve a number of research and engineering problems related to scalability and usability. Another avenue for research is to identify more meta-requirements that can be formalized and automatically analyzed, in addition to the three we considered in this paper. The more meta-requirements we have, the more errors in our requirements specifications we will automatically detect (at first), and the more *(mathematically founded)* trust in our requirements specification we will gain (at last).

References

1. J.-R. Abrial. Formal methods in industry: achievements, problems, future. *ICSE*, pages 761–768, 2006.
2. G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. *Formal Methods for the Design of Real-Time Systems*, pages 200–236, 2004.
3. A. G. Dahlstedt and A. Persson. Requirements interdependencies - moulding the state of research into a research agenda. In *REFSQ*, pages 71–80, 2003.
4. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, New York, 1999. ACM.
5. B. Han, D. Gates, and L. Levin. From language to time: A temporal expression anchorer. *TIME*, pages 196–203, June 2006.

6. M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency analysis of state-based requirements. In *IEEE Trans. on SW Engineering*, pages 3–14, 1995.
7. C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. SW Eng. and Meth.*, 5(3):231–261, 1996.
8. J. Hoenicke. *Combination of Processes, Data, and Time*. PhD thesis, University of Oldenburg, July 2006.
9. S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *ICSE '05: Proc. 27th Int. Conf. Softw. Eng.*, pages 372–381, New York, NY, USA, 2005. ACM.
10. T. Kuhn. Acerules: Executing rules in controlled natural language. In *Int. Conf. on Web Reasoning and Rule Systems*. Springer, 2007.
11. R. Meyer, J. Faber, J. Hoenicke, and A. Rybalchenko. Model checking duration calculus: a practical approach. *Formal Asp. Comput.*, 20(4-5):481–505, 2008.
12. A. Post, J. Hoenicke, and A. Podelski. rt-inconsistency: a new property for real-time requirements. In *FASE*, pages 34–49, 2011.
13. A. Post, J. Hoenicke, and A. Podelski. Vacuous real-time requirements. In *RE'11*. IEEE, 2011.
14. A. Post, I. Menzel, and A. Podelski. Applying restricted english grammar on automotive requirements — does it work? a case study. In *REFSQ*, pages 166–180, 2011.
15. J. Skakkebæk. Liveness and fairness in duration calculus. In B. Jonsson and J. Parrow, editors, *CONCUR '94*, volume 836, pages 283–298. Springer, 1994.
16. G. S. Walia and J. C. Carver. A systematic literature review to identify and classify software requirement errors. *Inf. Softw. Technol.*, 51(7):1087–1109, 2009.
17. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Acad. Pub., Norwell, USA, 2000.
18. L. Yu, S. Su, S. Luo, and Y. Su. Completeness and consistency analysis on requirements of distributed event-driven systems. In *TASE*, pages 241–244, Washington, 2008.
19. C. Zhou and M. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. Springer-Verlag, 2004.