DOI 10.1007/s00165-015-0348-9
BCS © 2015
Formal Aspects of Computing (2016) 28: 45–63

Formal Aspects of Computing

Analysing sanity of requirements for avionics systems

Jiří Barnat[1], Petr Bauch[1], Nikola Beneš[1], Luboš Brim[1], Jan Beran[2], Tomáš Kratochvíla[2]

[1] Faculty of Informatics, Masaryk University, Botanicka 68a, 602 00 Brno, Czech Republic
[2] Honeywell International, Aerospace, Advanced Technology Europe, Brno, Czech Republic

Abstract. In the last decade it became a common practice to formalise software requirements to improve the clarity of users' expectations. In this work we build on the fact that functional requirements can be expressed in temporal logic and we propose new sanity checking techniques that automatically detect flaws and suggest improvements of given requirements. Specifically, we describe and experimentally evaluate approaches to consistency and redundancy checking that identify all inconsistencies and pinpoint their exact source (the smallest inconsistent set). We further report on the experience obtained from employing the consistency and redundancy checking in an industrial environment. To complete the sanity checking we also describe a semi-automatic completeness evaluation that can assess the coverage of user requirements and suggest missing properties the user might have wanted to formulate. The usefulness of our completeness evaluation is demonstrated in a case study of an aeroplane control system.

Keywords: Requirement engineering, Linear temporal logic, Sanity checking

1. Introduction

The earliest stages of software development entail among others the activity of user requirements elicitation. The importance of clear specification of the requirements in the contract-based development process is apparent from the necessity of final-product compliance verification. Yet the specification itself is rarely described formally. Nevertheless, the formal description is an essential requirement for any kind of comprehensive verification. Recently, there have been tendencies to use the mathematical language of temporal logics, e.g. the *Linear Temporal Logic* (LTL), to specify functional system requirements. Restating requirements in a rigorous, formal way enables the requirement engineers to scrutinise their insight into the problem and allows for a considerably more thorough analysis of the final requirement documents [HJC+08].

Later in the development, when the requirements are given and a model is designed, the formal verification tools can provide a proof of correctness of the system being developed with respect to formally written requirements. The model of the system or even the system itself can be checked using model checking [BBČR10, CCG+02] or theorem proving [BFG+01] tools. If there are some requirements the system does not meet, the cause has to be found and the development reverted. The longer it takes to discover an error in the development, the more expensive the error is to mend. Consequently, errors made during requirements specification are among the most expensive ones in the whole development.

Correspondence and offprint requests to: J. Barnat, E-mail: barnat@fi.muni.cz

Model checking is particularly efficient in finding bugs in the design, however, it exhibits some shortcomings when it is applied to requirement analysis. In particular, if the system satisfies the formula then the model checking procedure only provides an affirmative answer and does not elaborate for the reason of the satisfaction. It could be the case that, e.g. the specified formula is a tautology, hence, it is satisfied for any system. Tautologies can be obvious, such as $p \vee \neg p$ or subtle, such as $p\ \mathcal{U} \neg(r \wedge q)$, where r stands for "The value of variable x is smaller than 5" and q stands for "The value of variable x is greater than 10". To mitigate the situation a subsidiary approach, the sanity checking, was proposed to check vacuity and coverage of requirements [Kup06]. Yet the existence of a model is still a prerequisite which postpones the verification until later phases in the development cycle.

The primary interest of this paper is to re-state and extend techniques of our own [BBB12a] that allow the developers of computer systems to check sanity of their requirements when it matters most, i.e. during the requirements stage. Sanity checking commonly consists of three parts: consistency checking, redundancy checking, and checking completeness of requirements. A consistent set of requirements is one that can be implemented in a single system. A vacuous requirement is satisfied trivially by a given system and redundancy is the equivalent of vacuity for the model-free case. Finally, a complete set of requirements extends to all sensible behaviours of a system. All three notions will be defined properly in Sect. 2.

In our previous work we redefined the notion of sanity checking of requirements written as LTL formulae and described its implementation and evaluation. The novelty of our approach we recapitulate and extend in this paper lies in the idea of liberating sanity checking from the necessity of having a model of the system under development. Our approach to consistency and vacuity checking presented before identified all inconsistent and vacuous subsets of the input set of requirements. This considerably simplified the work of requirement engineers because it allowed to pinpoint all the sources of inconsistencies. As for completeness checking, we proposed a new behaviour-based coverage metric. Assuming that the user specifies what behaviour of the system is sensible, our coverage metric calculated what portion of this behaviour is described by the requirements specifying the system itself. The method further suggested new requirements to the user that would have improved the coverage and thus ensured more accurate description of users' expectations.

Contribution The novelty of our consistency (redundancy) checking is that they produce all inconsistent (redundant) sets instead of a yes/no answer and their efficiency is demonstrated in an experimental evaluation. Finally, the completeness checking presents a new behaviour-based coverage and suggests formulae that would improve the coverage and, consequently, the rigour of the final requirements.

In this paper we extend [BBB12a] with a modified workflow that allows requirement engineers to explicitly specify universal ($\forall$) and existential ($\exists$) nature of individual requirements allowing thus to more naturally express some of the system properties. We also evaluate the robustness and sensitivity of our consistency and completeness checking procedures with respect to the use of different LTL to BA translators. Furthermore, we adopt vacuity checking for individual LTL formulae into the framework and add proper references to relevant previous work. Finally, we report on experience we gathered in cooperation with Honeywell International when applying our methodology on some real-life industrial use cases.

1.1. Related work

The use of model checking with properties (specified in CTL) derived from real-life avionics software specifications was successfully demonstrated in [CAB+98]. This paper intends to present a preliminary to such a use of a model checking tool, because there the authors presupposed sanity of their formulae. The idea of using coverage as a metric for completeness can be traced back to software testing, where it is possible to use LTL requirements as one of the coverage metrics [WRHM06, RWH07].

Model-based sanity checking was studied thoroughly and using various approaches, but it is intrinsically different from model-free checking presented in this paper. Completeness is measured using metrics based on the state space coverage of the underlying model [CKKV01, CKV01]. Vacuity of temporal formulae was identified as a problem related to model checking and solutions were proposed in [KV03] and in [Kup06], again expecting existence of a model.

Checking consistency (or satisfiability) of temporal formulae is a well understood problem solved using various techniques in many papers (most recently using SAT-based approach in [RDB+05] or in [RV07] where it was used as a comparison between different LTL to Büchi translation techniques). The classical problem is formulated as to decide whether a set of formulae is internally consistent. In this paper, however, a more elaborate answer

is sought: specifically which of the formulae cause the inconsistency. The approach is then extended to vacuity which is rarely used in model-free sanity checking.

We have adopted the idea of searching for the smallest inconsistent subset of a set of requirements from the SAT community, where finding the minimal unsatisfiable core (a set of propositional clauses) is an important feature of SAT solvers, see e.g. [LMS04]. The authors of [LS08] extend the approach to finding all minimal unsatisfiable subsets, again in the context of propositional logic. Their exclusion of checking those subsets containing known unsatisfiable subsets was extended by our algorithm with a dual heuristic: exclude also subsets with known satisfiable supersets.

A problem related to consistency is that of *realisability* [ALW89]. Realisability generalises consistency by allowing some of the atomic propositions to be controlled by the environment. A set of system requirements is then realisable if there is a strategy which can react to arbitrary environment choices while satisfying the requirements. If no propositions are controlled by the environment then the notion of realisability becomes equivalent to consistency. A number of tools implement realisability, RATSY [BCG+10] for example, but again limit their responses to stating whether the requirements are realisable as a whole. Thus, even though the theory of minimal unrealisable cores had been known for some time [CRST08, KHB09], searching for smallest inconsistent subsets is a feature unique for our tool. We would also like to point out the theory described in [Sch12] which extends the unrealisability cores to search even within individual formulae, providing even finer information on the quality of requirements.

Completeness of formal requirements is not as well-established and its definition often differs. The most thorough research in algorithmic evaluation of completeness was conducted in [HL95, Lev00, MTH03]. Authors of those papers use RSML (Requirements State Machine Language) to specify requirements which they translate (in the last paper) to CTL and to a theorem prover language to discover inconsistencies and detect incompleteness. Their notion of completeness is based on verifying that for every input value there is a reaction described in the specification. This paper presents completeness as considering all behaviours described as sensible (and either refuting or requiring them). Finally, a novel semi-formal methodology is proposed in this paper, that recommends new requirements to the user, that have the potential to improve completeness of the input specification.

Another approach at defining and checking completeness was pursued in [KGG99] in terms of evaluating the extend to which an implementation describes the intended specification. Their evaluation is based on the simulation preorder and searching for unimplemented transition/state evidence. In order to create a metric that would drive the engine for generating new requirements we have build an evaluation algorithm that enumerates the similarity of two paths. Our notion of similarity is to certain extend based on unimplemented transitions, but we carry the enumeration from two transition up to the whole automata, thus finally obtaining our completeness metric. The authors of [BB09] elaborate a method for measuring the quality of a set of requirements by detecting the presence of such a subsets, that together require behaviour that could not be detected by individual requirements. Similarly as in [HL95], the interesting behaviour is described in terms of reactions to input values observed at discrete time instances, yet the properties are limited to those of the form $\forall\, t,\, A_t(t) \Rightarrow Z_t(t)$.

Finally, the notion of coverage is well-established in the area of software testing. The similarity with our work and that described for example in [RLS+03, FG03] lies in that both approaches attempt to generate a description of sets of behaviours that remain uncovered. The dissimilarity lies in that we describe these uncovered sets in term of software requirements (or LTL formulae), whereas automated testing coverage produces collections of test, i.e. individual runs of a particular system.

2. Preliminaries

This section serves as a motivation for and a reminder of the model checking process and its connection to sanity checking. A knowledgeable reader might find it slow-paced and cursory, but its primary function is to justify the use of formal specifications and, as such, requires more compliant approach.

2.1. LTL model checking

Definition 1 Let AP be the set of *atomic propositions*. Then the recursive definition below specifies all well-formed *LTL formulae* over AP, where $p \in AP$:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathcal{X}\varphi \mid \varphi\, \mathcal{U}\, \varphi$$

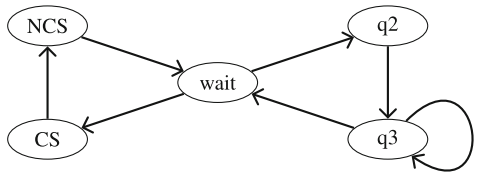

Fig. 1. LTS for Peterson's mutual exclusion protocol (with only one process) and two liveness LTL formulae

Example 1 There are some well-established syntactic simplifications of the LTL language, e.g. $false := p \wedge \neg p$, $true := \neg false$, $\varphi \Rightarrow \psi := \neg(\varphi \wedge \neg\psi)$, $\mathcal{F}\varphi := true\, \mathcal{U}\, \varphi$, $\mathcal{G}\varphi := \neg(\mathcal{F}\neg\varphi)$. Assuming that $AP = \{\alpha := (c = 5), \beta := (a \neq b)\}$, these are examples of well-formed LTL formulae: $\mathcal{G}\beta, \alpha\, \mathcal{U} \neg\beta$. △

In classical model checking one usually verifies that a model of the system in question satisfies the given set of LTL-specified requirements. That is not possible in the context of this paper because in the requirements stage there is no model to work with. Nevertheless, to better understand the background of LTL model checking let us assume that the system is modelled as a *Labelled Transition System* (LTS).

Definition 2 Let Σ be a set of state labels (it will mostly hold that $\Sigma = AP$). Then an *LTS* $M = (S, \rightarrow, \nu, S_0)$ is a tuple, where: S is a set of states, $\rightarrow \subseteq S \times S$ is a transition relation, $\nu : S \rightarrow 2^{\Sigma}$ is a valuation function and $S_0 \subseteq S$ is a nonempty set of initial states. A function $r : \mathbb{N} \rightarrow S$ is an *infinite run* over the states of M if $r(0) \in S_0, \forall\, i : r(i) \rightarrow r(i+1)$. The *trace* or *word* of a run is a function $w : \mathbb{N} \rightarrow 2^{\Sigma}$, where $w(i) = \nu(r(i))$.

An LTL formula states a property pertaining to an infinite trace (a trace that does not have to be associated with a run). Assuming an LTS to be a model of a computer program then a trace represents one specific execution of the program. Also the infiniteness of the executions is not necessarily an error—programs such as operating systems or controlling protocols are not supposed to terminate.

Definition 3 Let w be an infinite word and let φ be an LTL formula over Σ. Then it is possible to decide if w satisfies φ, $w \models \varphi$, based on the following rules:

$$
\begin{array}{ll}
w \models p & \textit{iff } p \in w(0), \\
w \models \neg\varphi & \textit{iff } w \not\models \varphi, \\
w \models \varphi_1 \wedge \varphi_2 & \textit{iff } w \models \varphi_1 \textit{ and } w \models \varphi_2, \\
w \models \mathcal{X}\varphi & \textit{iff } w_1 \models \varphi, \\
w \models \varphi_1\, \mathcal{U}\, \varphi_2 & \textit{iff } \exists\, i. \forall\, j < i : w_j \models \varphi_1, w_i \models \varphi_2,
\end{array}
$$

where $w(i)$ is the i-th letter of w and w_i is the i-th suffix of w.

Example 2 Figure 1 contains LTS for a process engaged in Peterson's mutual exclusion protocol. The protocol can control access to the critical section (state *CS*) for arbitrarily many processes that communicate using global variables to determine which process will be granted access next. The two LTL formulae verify the liveness property of the protocol:

1: If a process *waits* for an access to the critical section it will eventually get there.
2: A process outside the critical section will eventually get inside, and this hold in any state of the system. △

Traditionally, a system as a whole is considered to satisfy an LTL formula if all its executions (all infinite words over the states of its LTS) do. However, we might occasionally be also interested in the question whether at least one of a system's executions satisfies the given LTL formula. We thus suggest explicit path quantification using the $\forall$ and $\exists$ quantifiers as follows: For a system model M and a formula φ, we write $M \models \forall\, \varphi$ if all executions of M satisfy φ, and $M \models \exists \varphi$ if there is at least one execution of M satisfying φ. We call the former kind of path-quantified formulae *universal* and the latter kind *existential*.

There are several notes to be made. First, we do not allow for nesting of path-quantified formulae as we want to avoid the complexity of dealing with full CTL*. We only occasionally use the negation operator with the meaning $\neg \forall\, \varphi \equiv \exists \neg\varphi$ and vice versa. Second, one of the advantages of making the hidden universal quantification of LTL more explicit is to reduce the confusion over the fact that LTL seemingly does not follow the law of excluded middle: We can have a system model that satisfies neither φ nor $\neg\varphi$. Third, using path-quantified LTL formulae is really just a convenience, as it does not make the verification task any harder. Indeed, to verify whether a system model satisfies $\exists\, \varphi$ is the same as to verify whether the system model satisfies $\forall\, \neg\varphi$ and invert the answer. Efficient

verification of that satisfaction, however, requires a more systematic approach than enumeration of all executions. An example of a successful approach is the enumerative approach using *Büchi automata*.

Definition 4 A *Büchi automaton* is a pair $A = (M, F)$, where M is an LTS and $F \subseteq S$. An automaton A accepts an infinite word w ($w \in L(A)$) if there exists a run r for w in M and there is a state from F that appears infinitely often on r, i.e. $\forall\, i\, \exists j > i : r(j) \in F$.

Arbitrary LTL formula φ can be transformed into a Büchi automaton A_φ such that $w \models \varphi \Leftrightarrow w \in L(A_\varphi)$. Also checking that every execution satisfies φ is equivalent to checking that no execution satisfies $\neg\varphi$. It only remains to combine the LTS model of the given system M with $A_{\neg\varphi}$ in such a way that the resulting automaton will accept exactly those words of M that violate φ. Finally, deciding existence of such a word—and by extension verifying correctness of the system—has been shown equivalent to finding accepting cycle in a graph.

In the following sections we shall usually deal with the standard approach to LTL first, i.e. consider implicit universal path quantification of the formulae. We then follow with the extension to existential path-quantified formulae.

2.2. Model-based sanity checking

As described in the introduction the model checking procedure is not designed to decide why was a certain property satisfied in a given system. That is a problem, however, because the reasons for satisfaction might be the wrong ones. If for example the system is modelled erroneously or the formula is not appropriate for the system then it is still possible to receive a positive answer.

These kinds of inconsistencies between the model and the formula are detected using sanity checking techniques, namely *vacuity* and *coverage*. In this paper they will be described for comparison with their model-free versions; interested reader should consult for example [Kup06] for more details.

Let K be an LTS, φ a formula and ψ its subformula. Let further $\varphi[\psi'/\psi]$ be the modified formula φ in which its subformula ψ is substituted by ψ'. Then ψ *does not affect the truth value of* φ *in* K if the following property holds: K satisfies $\varphi[\psi'/\psi]$ for every formula ψ' iff K satisfies φ. Then a system K *satisfies a formula* φ *vacuously* if K satisfies φ and there is a subformula ψ of φ such that ψ does not affect φ in K.

A state s of an LTS K is q*-covered by* φ, for a formula φ and an atomic proposition q, if K satisfies φ but $\tilde{K}_{s,q}$ does not satisfy φ. There $\tilde{K}_{s,q}$ stands for an LTS equivalent to K except the valuation of q in the state s is flipped.

If one could extract the underlying ideas of vacuity and coverage and show that they do not necessarily require the existence of a model, it would be possible to extrapolate these notions into the model-free environment. Vacuity states that the satisfaction of a formula is given extrinsically (by *an* environment) and is not related to the formula itself. Thus we may consider the remaining formulae from the set of requirements to constitute the environment. Coverage, on the other hand, attempts to capture the amount of system behaviour that is described by the formulae. Again, we can use the remaining formulae to form the system, but in order to establish a coverage metric capable of differentiating various formulae, we will need to extend the notion from state-coverage to path-coverage.

Although these notions of sanity are dependent on the system that is being verified, we might still utilise at least the notion of vacuity in a model-less setting, using a concept of *vacuity witnesses*. A vacuity witness to a given formula φ is a formula ψ with the following property: If a system K satisfies ψ then it satisfies φ vacuously. A standard example is the request-response formula $\mathcal{G}(req \Rightarrow \mathcal{F}\, resp)$ "every request is followed by a response", whose two vacuity witnesses are $\mathcal{G}(\neg req)$ "no request ever happens" and $\mathcal{G}\,\mathcal{F}(resp)$ "responses happen infinitely often". The vacuity witnesses for a given formula can be automatically generated, see [BBDER01]. Moreover, as it is desired for the vacuity witnesses not to hold, we may include this requirement in our original set of requirements with the help of path-quantified LTL formulae. For example, if our set of requirements contains the formula $\forall\, \mathcal{G}(req \Rightarrow \mathcal{F}\, resp)$, we add the formulae $\exists\, \mathcal{F}(req)$ and $\exists\, \mathcal{F}\,\mathcal{G}(\neg resp)$ to the set—those are the negations of the vacuity witnesses.

In the rest of this paper, we translate the concepts of model-based sanity into model-free environment. However, to avoid confusion, we use the word *redundancy* instead of model-less vacuity.[1] We further supplement these notions with consistency verification thus arriving at a method that would aid the creation of a reasonable set of requirements, i.e. a set such that it is reasonable to ask if a system satisfies it or not. If, however, the intention is

[1] This is in contrast to the original paper [BBB12a] where *redundancy* has been called *vacuity* as the paper did not include vacuity witnesses.

to build a system based on this set, then a further property of such set would be desirable and that the set completely describes the system under development. Consequently, a *sane* set of requirements is consistent, without redundancies, and complete in a sense we will formalise in Sect. 4.

3. Checking consistency and redundancy

As various studies concluded, undetected errors made early in the development cycle are the most expensive to eradicate. Thus it is very important that the outcome of the requirements stage—a database of well-formed, traceable requirements—is what the customer intended and that nothing was omitted (not even unintentionally). While a procedure that would ensure the two properties cannot be automated, this paper proposes a methodology to check the sanity of requirements. In the following the *sanity checking* will be considered to consist of 3 related tasks: *consistency*, *redundancy* and *completeness* checking. As consistency and redundancy are more closely related, we focus on them in this section and dedicate the following section solely to completeness.

Definition 5 A set Γ of LTL formulae over AP is *consistent* if $\exists\, w \in AP^\omega : w$ satisfies $\bigwedge \Gamma$. Checking consistency of a set Γ entails finding all minimal inconsistent subsets of Γ. A formula φ is *redundant* with respect to a set of formulae Γ if $\bigwedge \Gamma \Rightarrow \varphi$. To check redundancy of a set Γ entails finding all pairs of $\langle \varphi \in \Gamma, \Phi \subseteq \Gamma \rangle$ such that Φ is consistent and $\Phi \Rightarrow \varphi$ (and for no $\Phi' \subseteq \Phi$ does it hold that $\Phi' \Rightarrow \varphi$).

The existence of the appropriate w can be tested by constructing $A_{\bigwedge \Gamma}$ and checking that $L(A_{\bigwedge \Gamma})$ is non-empty. The procedure is effectively equivalent to model checking where the model is a clique over the graph with one vertex for every element of 2^{AP} (allowing every possible behaviour).

This approach to consistency and redundancy is especially efficient if a non-trivial set of requirements needs to be processed and standard sanity checking would only reveal if there is an inconsistency (or redundancy) but would not be able to locate the source. Furthermore, dealing with larger sets of requirements entails the possibility that there will be several inconsistent subsets or that a formula is redundant due to multiple small subsets. Each of these conflicting subsets needs to be considered separately which can be facilitated using the methodology proposed in this paper.

Example 3 Let us assume that there are five requirements formalised as LTL formulae over a set of atomic propositions (two-valued signals) $\{p, q, a\}$. They are

1. Eventually, the presence of signal p will entail its continued presence until the signal q is observed. $\varphi_1 = \mathcal{F}(p \Rightarrow p\, \mathcal{U}\, q)$
2. The signal p will occur infinitely often. $\varphi_2 = \mathcal{G}\mathcal{F}p$
3. The signals a and p cannot occur at the same time. $\varphi_3 = \mathcal{G}\neg(a \wedge p)$
4. Each occurrence of the signal q requires that signal a was observed in the previous time step. $\varphi_4 = \mathcal{G}(\mathcal{X}q \Rightarrow a)$
5. The signal q will occur infinitely often. $\varphi_5 = \mathcal{G}\mathcal{F}q$

In this set the formula φ_4 is inconsistent due to the first 3 formulae and the last formula is redundant with respect to (implied by) the first 2 formulae. △

We now reformulate the notions of Definition 5 to extend also to path-quantified LTL formulae. Again, we are interested in finding the minimal inconsistent subsets and minimal redundancy witnesses.

Definition 6 A set Γ of path-quantified LTL formulae over AP is *consistent* if there is a system model M such that $M \models \varphi$ for all $\varphi \in \Gamma$. A path-quantified formula φ is *redundant* with respect to a set of formulae Γ if every model M satisfying all formulae of Γ also satisfies φ.

We now show that checking consistency and redundancy in the path-quantified setting can be easily reduced to the case in the standard setting. Let us partition the set Γ into existential formulae $\exists\Gamma$ and universal formulae $\forall\Gamma$, i.e. $\exists\Gamma := \Gamma \cap \{\exists\varphi\}$, for φ quantifier-free, and $\forall\Gamma := \Gamma \setminus \exists\Gamma$. Further, let $\Psi 1\exists := \{\Phi \cup \{\varphi\} \mid \Phi \subseteq \forall\Gamma, \varphi \in \exists\Gamma\}$.

Theorem 1 In order to locate all smallest inconsistent subsets of Γ is suffices to search among $\Psi 1\exists$. Similarly, checking redundancy may correctly be limited to checking among pairs

$\langle \varphi \in \Gamma, \Phi \subseteq \forall\Gamma \rangle$ φ universal
$\langle \forall\psi, \Phi \in \Psi 1\exists \cup \forall\Gamma \rangle$ φ existential, $\varphi = \exists\psi$.

Proof Let us first consider consistency. Clearly, if Γ only consists of universal formulae, we may simply ignore their quantifiers. In the case of Γ containing existential formulae, we can make the following two observations:

- First observation: Two existential formulae are always consistent provided that they are both self-consistent, i.e. none of them is equal to *false*. This can be easily seen as we can always provide a model with exactly two paths, each satisfying one of the formulae. This observations tells us that every minimal inconsistent subset contains at most one existential formula.
- Second observation: Let us write $\Gamma^{\forall}$ to denote the set Γ in which all path quantifiers have been changed to $\forall$. If the set Γ contains at most one existential formula then the following holds: Γ is consistent iff $\Gamma^{\forall}$ is consistent. One direction is obvious, the other follows from the fact that we can always provide a single-path model as a proof of consistency of $\Gamma^{\forall}$.

The result of the two observations is that if we are careful to never consider sets with more than one existential formula, we may freely ignore the path-quantifiers.

Let us now consider redundancy. It is clear that the following holds: φ is redundant with respect to Γ iff $\Gamma \cup \{\neg\varphi\}$ is inconsistent. This means that redundancy checking is easily reduced to consistency checking. This fact holds in the path-quantified setting as well as in the standard one and will be used later in the implementation. Note that the first observation above has the following consequences: If φ is a universal formula, then it makes sense to consider Γ with universal formulae only, as $\neg\varphi$ is existential. Furthermore, if φ is an existential formula, it makes sense to consider Γ containing at most one existential formula. □

From the above theorem it follows that all formulae that require consideration are universal, which is the implicit quantifier of quantifier-free formulae. In other words the two statements above enable the same algorithms for checking sanity, described in the next section, to be used in the path-quantified setting as well. Yet the validity of the two statements requires a more detailed argumentation. Before we come to the implementation, let us illustrate the interplay between the concepts discussed so far (model-less consistency, redundancy, and vacuity witnesses) on a more comprehensive example.

Example 4 Consider a set of requirements containing just two items:

1. If a signal a is ever activated, it will remain active indefinitely, i.e. in every time step it holds that if a is active now it will also be active in the next time step. Translating this natural language requirement to LTL we arrive at: $\varphi_1 = \mathcal{G}(a \Rightarrow \mathcal{X}\, a)$
2. If a signal a is ever activated, it will not be active in the next time step, i.e. a must never hold in two successive steps. After translation the appropriate LTL formula thus stands: $\varphi_2 = \mathcal{G}(a \Rightarrow \mathcal{X}\neg a)$

Note that although such requirements might seem inconsistent at first glance, they are, in fact, consistent as any consistency checking algorithm could demonstrate. What the following redundancy analysis is about to reveal is that, while consistent, it is not reasonable to require both formulae to hold at the same time.

Let us elaborate and formalise the reasoning step of redundancy checking. We begin by constructing the vacuity witnesses according to the method in [BBDER01]. The vacuity witnesses for φ_1 are: $\mathcal{G}(\neg a)$, $\mathcal{G}\,\mathcal{X}\, a$. The first witness is quite intuitive, if the signal a is never activated in a system then requiring φ_1 to hold is unreasonable. Similarly, if at every time step we have that a will be active in the next step then clearly the presence of a in this time step is irrelevant. Exactly the same kind of mechanisable reasoning leads to the vacuity witnesses for φ_2: $\mathcal{G}(\neg a)$, $\mathcal{G}\,\mathcal{X}(\neg a)$. We thus create the new requirements as described in the previous section. The set of requirements now contains $\forall\,\varphi_1$, $\forall\,\varphi_2$, $\exists\,\mathcal{F}\, a$, $\exists\,\mathcal{F}\,\mathcal{X}(\neg a)$, $\exists\,\mathcal{F}\,\mathcal{X}\, a$. Take for example $\exists\,\mathcal{F}\,\mathcal{X}(\neg a)$, resulting from the negated witness $\mathcal{G}\,\mathcal{X}\, a$. Being existentially quantified, this requirement demands the existence of at least one computation of the system, one on which a state whose successor does not observe a will eventually occur. Note that we have converted the implicit formulae into universal ones and ignored the duplicity of the vacuity witnesses.

We now check the redundancy of the requirements and discover that the formula $\exists\,\mathcal{F}\,\mathcal{X}\, a$ implies $\exists\,\mathcal{F}\, a$, we thus drop $\exists\,\mathcal{F}\, a$ from the set. The remaining four formulae are checked for consistency. We find $\{\forall\,\varphi_1, \forall\,\varphi_2, \exists\,\mathcal{F}\,\mathcal{X}\, a\}$ as the only minimal inconsistent subset. The conclusion is that although φ_1 and φ_2 are consistent, the only models that satisfy both of them satisfy them redundantly. The two formulae can thus be said to be redundantly consistent. △

Algorithm 1: Consistency Check

```
1 while t ← getTask() do
2     if t.checked() then
3         genSuccs(t)
4     else
5         t.checked ← verCons(t)
6         updateSets(t)
7         Pool.enqueue(t)
```

Algorithm 2: genSuccs(*Task* t)

```
1 if t.con() then
2     if t.dir =↑ then
3         genSupsets(t)
4 else
5     if t.dir =↓ then
6         genSubsets.(t)
```

3.1. Implementation of consistency checking

Let us henceforth denote one specific instance of consistency (or redundancy) checking as a *check*. For consistency and a set Γ it means to check that for some $\gamma \subseteq \Gamma$ is $\bigwedge \gamma$ satisfiable. For redundancy it means for $\gamma \subseteq \Gamma$ and $\varphi \in \Gamma$ to check that $\bigwedge \gamma \Rightarrow \varphi$ is satisfiable. In the worst case both consistency and redundancy checking would require an exponential number of checks. However, the proposed algorithm considers previous results and only performs the checks that need to be tested.

Both consistency and redundancy checking use three data structures that facilitate the computation. First, there is the queue of verification tasks called *Pool*, then there are two sets, *Con* and *Incon*, which store the consistent and inconsistent combinations found so far. Finally, each individual *task* contains a set of integers (that uniquely identifies formulae from Γ) and a *flag* value (containing three bits for three binary properties). First, whether the satisfaction check was already performed or not. Second, if the combination is consistent. And the third bit specifies the direction in subset relation (up or down in the Hasse diagram) in which the algorithm will continue. The successors will be either subsets or supersets of the current combination.

The idea behind consistency checking is very simple (listed as Algorithm 1). The pool contains all the tasks to be performed and these tasks are of two types: either to check consistency of the combination or to generate successors. The symmetry of the solution allows for parallel processing (multiple threads performing the Algorithm 1 at the same time) given that the data structures are properly protected from race conditions. The pool needs to be initialised with all single element subsets of Γ and Γ itself, thus in the subsequent iteration will be checked the supersets of the former and subsets of the latter.

Algorithm 2 is called when the task t on the top of *Pool* is already checked. At this point either all subsets or all supersets of t should be enqueued as tasks. But not all successors need to be inspected, e.g. if t is consistent then also all its subsets will be consistent—that is clearly true and no subset of t needs to be checked.

That observation is utilised again in Algorithm 3. It does not suffice to stop generating subsets and supersets when its immediate predecessors are found consistent (inconsistent), because it can also happen that the combination to be checked was formed in a different branch of the Hasse diagram of the subset relation. In order to prevent redundant satisfiability checks two sets are maintained *Con* and *Incon* (see how these are used on line 4 of Algorithm 3).

Algorithm 3: genSupsets(*Task* t)

```
1 foreach i ∈ {1,...,n} do
2     t.add(i)
3     if ∀X ∈ Con : X ⊉ t ∧
4        ∀X ∈ Incon : X ⊈ t then
5         enqueue(t)
6     t.erase(i)
```

Algorithm 4: verCons(t=⟨$i_1,\ldots,i_j$⟩)

```
1 F ← createConj(φ_{i_1},...,φ_{i_j})
2 A ← transform2BA(F)
3 return findAccCycle(A)
```

The actual consistency (and quite similarly also redundancy) checking is less complicated (see Algorithm 4) and may even be delegated to a third party tool. First, the conjunction of formulae encoded in the task is created. In our setting we then check the appropriate Büchi automaton for the existence of an accepting cycle: using nested DFS [CVWY92]. Hence the Algorithm 4 can easily be substituted with another method for checking consistency

of a set of LTL formulae. A realisability checking tool, such as RATSY, could also be applied: one only needs to state that all signals are controlled by the system.

3.1.1. Extension to redundancy checking

The only difference when performing the redundancy checking is that the task t consists of a list $\langle i_1, \ldots, i_j \rangle$ which can be empty, and one index i_k. Since the task is to decide whether $\varphi_{i_1} \wedge \ldots \wedge \varphi_{i_j} \Rightarrow \varphi_{i_k}$ the line 1 needs to be altered to:
$\mathsf{F} \leftarrow \texttt{createConj}(\varphi_{i_1}, \ldots, \varphi_{i_j}, \neg\varphi_{i_k})$
This is due to the fact discussed in the previous section, namely that the satisfiability of $\varphi_{i_1} \wedge \ldots \wedge \varphi_{i_j} \wedge \neg\varphi_{i_k}$ implies $\varphi_{i_1} \wedge \ldots \wedge \varphi_{i_j} \not\Rightarrow \varphi_{i_k}$, i.e. φ_{i_k} is not redundant with respect to $\{\varphi_{i_1}, \ldots, \varphi_{i_j}\}$.

Discarding the Büchi automata in every iteration may seem unnecessarily wasteful, especially since synchronous composition of two (and more) automata is a feasible operation. However, the size of an automaton created by composition is a multiplication of the sizes of the automata being composed. Furthermore, it would not be possible to use the size optimising techniques employed in LTL to Büchi translation. And these techniques work particularly well in our case, because the translated formulae (conjunctions of requirements) have relatively small nesting depth (maximal depth among requirements +1).

Example 5 We will demonstrate the search for smallest inconsistent subsets on the following set of requirements.

1. A request to increase the cabin temperature a may always be issued and each such request will eventually be granted by heating unit b. $\varphi_1 = \mathcal{G}(\mathcal{F}\, a \wedge (a \Rightarrow \mathcal{F}\, b))$
2. There is only a finite amount of fuel for the heating unit. $\varphi_2 = \mathcal{F}\,\mathcal{G}(\neg b)$
3. The system must be robust to handle overriding command c after which no request to increase the temperature may be issued. $\varphi_3 = \mathcal{F}(c \wedge c \Rightarrow \mathcal{G}\neg a)$
4. There will be no request to increase the temperature initially, but every heating will be preceded by a request which must be turned off after one time unit once the heating starts. Also the request is always issued until the heating starts. $\varphi_4 = \mathcal{G}(a\, \mathcal{U}\, b \wedge \mathcal{X}\, b \Rightarrow (a \wedge \mathcal{X}\, a \wedge \mathcal{X}\mathcal{X}\neg a)) \wedge \neg a$

Our consistency checking algorithm begins by checking in parallel the individual requirements $\varphi_1, \ldots, \varphi_4$ for consistency and also the whole set $\Gamma = \{\varphi_1, \ldots, \varphi_4\}$. It discovers that φ_4 is inconsistent on its own and that Γ is also inconsistent. The fact that φ_4 is inconsistent is employed by not considering any of the sets containing φ_4, i.e. $\{\varphi_1, \varphi_4\}$, $\{\varphi_2, \varphi_4\}$, $\{\varphi_3, \varphi_4\}$, $\{\varphi_1, \varphi_2, \varphi_4\}$, $\{\varphi_1, \varphi_3, \varphi_4\}$, and $\{\varphi_2, \varphi_3, \varphi_4\}$. Next, the algorithm checks the supersets of the largest consistent sets, i.e. $\{\varphi_1, \varphi_2\}$, $\{\varphi_1, \varphi_3\}$, and $\{\varphi_2, \varphi_3\}$. Discovering that both $\{\varphi_1, \varphi_2\}$ and $\{\varphi_1, \varphi_3\}$ are inconsistent it terminates since $\{\varphi_1, \varphi_2, \varphi_3\}$ is necessarily also inconsistent. The smallest inconsistent subsets are thus $\{\varphi_4\}$, $\{\varphi_1, \varphi_2\}$, and $\{\varphi_1, \varphi_3\}$. △

4. Checking completeness

The completeness checking is a little more involved: this is in fact the part that provably cannot be fully automated. Hence the paper will first describe the problem and then detail the semi-automatic solution proposed.

Let us assume that the user specifies three types of requirements: environmental assumptions Γ_A, required behaviour Γ_R and forbidden behaviour Γ_F. The environmental assumptions represent the sensible properties of the world a given system is to work in, e.g. "The plane is either in the air or on the ground, but never in both these states at once", in LTL this would read $\mathcal{G}(ground \Leftrightarrow \neg air)$. The required behaviour represents the functional requirements imposed on the system: the system will not be correct if any of these is not satisfied, e.g. "If the plane is ever to fly it must first be tested on the ground": $\mathcal{F}\, air \Rightarrow (ground \wedge tested)$. Dually, the forbidden behaviour contains those patterns that the system must not display, e.g. "Once tested the plane should be put into air within the next three time steps": $\neg\ tested\ \mathcal{U}(air \vee \mathcal{X}\, air \vee \mathcal{X}\mathcal{X}\, air)$.

Assume henceforth the following simplifying notation for Büchi automata: let f be a propositional formula over capital Latin letters $A, B, \ldots$ barred letters $\bar{A}, \bar{B}, \ldots$ and Greek letters $\alpha, \beta, \ldots$, where A substitutes $\bigwedge_{\gamma\in\Gamma_A} \gamma$, $\bar{A}$ stands for $\bigvee_{\gamma\in\Gamma_A} \gamma$ and all Greek letters represent simple LTL formulae. Then $\mathcal{A}_f$ denotes such a Büchi automaton that accepts all words satisfying the substituted f, e.g. $\mathcal{A}_{A\vee\bar{B}\wedge\varphi}$ accepts words satisfying $\bigwedge_{\gamma\in\Gamma_A} \gamma \vee \bigvee_{\gamma\in\Gamma_B} \gamma \wedge \varphi$. The automaton $\mathcal{A}_A$ thus describes the part of the state space the user is interested in and which the required and forbidden behaviour should together submerge. That most commonly is not the case with freshly elicited

requirements and therefore the problem is the following: find sufficiently simple formulae over AP that would, together with formulae for R and F, cover a large portion of $\mathcal{A}_A$. In other words to find such φ that $\mathcal{A}_{R \vee \bar{F} \vee \varphi}$ covers as much of $\mathcal{A}_A$ as possible.

In order to evaluate the size of the part of $\mathcal{A}_A$ covered by a single formula, i.e. how *much* of the possible behaviour is described by it, an evaluation methodology for Büchi automata needs to be established. The plain enumeration of all possible words accepted by an automaton is impractical given the fact that Büchi automata operate over infinite words. Similarly, the standard completeness metrics based on state coverage [CKV01, TK01] are unsuitable because they do not allow for comparison of sets of formulae and they require the underlying model. Equally inappropriate is to inspect only the underlying directed graph because Büchi automata for different formulae may have isomorphic underlying graphs.

The methodology proposed in this paper is based on the notion of *almost-simple* paths and *directed partial coverage* function.

Definition 7 Let G be a directed graph. A *path* π in G is a sequence of vertices $v_1, \ldots, v_n$ such that $\forall\, i : (v_i, v_{i+1})$ is an edge in G. A path is *almost-simple* if no vertex appears on the path more than twice. The notion of almost-simplicity is also applicable to words in the case of Büchi automata.

With almost-simple paths one can enumerate the behavioural patterns of a Büchi automaton without having to handle infinite paths. Clearly, it is a heuristic approach and a considerable amount of information will be lost but since all simple cycles will be considered (and thus all patterns of the infinite behaviour) the resulting evaluation should provide sufficient distinguishing capacity (as demonstrated in Sect. 5.2).

Knowing which paths are interesting it is possible to propose a methodology that would allow comparing two paths. There is, however, a difference between Büchi automata that represent a computer system and those built using only LTL formulae (ones that should restrict the behaviour of the system). The latter automata use a different evaluation function $\hat{\nu}$ that assigns to every edge a set of literals. The reason behind this is that the LTL-based automaton only allows those edges (in the system automaton) for which their source vertex has an evaluation compatible with the edge evaluation (now in the LTL automaton).

Definition 8 Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two (LTL) Büchi automata over AP and let AP_L be the set of literals over AP. The *directed partial coverage* function Λ assigns to every pair of edge evaluations a rational number between 0 and 1, $\Lambda : 2^{AP_L} \times 2^{AP_L} \to \mathbb{Q}$. The evaluation works as follows

$$\Lambda(A_1 = \{l_{11}, \ldots, l_{1n}\}, A_2 = \{l_{21}, \ldots, l_{2m}\}) = \begin{cases} 0 & \exists\, i, j : l_{1i} \equiv \neg l_{2j} \\ p/m & \text{otherwise} \end{cases}$$

where $p = \mid A_1 \cap A_2 \mid$.

From this definition one can observe that Λ is not symmetric. This is intentional because the goal is to evaluate how much a path in $\mathcal{A}_2$ covers a path in $\mathcal{A}_1$. Hence the fact that there are some additional restricting literals on an edge of $\mathcal{A}_1$ does not prevent automaton $\mathcal{A}_2$ to display the required behaviour (the one observed in $\mathcal{A}_1$).

The extension of coverage from edges to paths and to automata is based on averaging over almost-simple paths. An almost-simple path π_2 of automaton $\mathcal{A}_2$ covers an almost-simple path π_1 of automaton $\mathcal{A}_1$ by $\Lambda(\pi_1, \pi_2) = \frac{\sum_{i=0}^{n} \Lambda(A_{1i}, A_{2i})}{n}$ per cent, where n is the number of edges and A_{ji} is the set of labels on i-th edge on π_j. Then automaton $\mathcal{A}_2$ covers $\mathcal{A}_1$ by $\Lambda(\mathcal{A}_1, \mathcal{A}_2) = \frac{\sum_{i=0}^{m} \max_{2i} \Lambda(\pi_{1i}, \pi_{2i})}{m}$ per cent, where m is the number of almost-simple paths of $\mathcal{A}_1$ that end in an accepting vertex. It follows that coverage of 100 % occurs when for every almost-simple path of one automaton there is an almost-simple path in the other automaton that exhibits similar behaviour.

4.1. Implementation of completeness checking

The high-level overview of the implementation of the proposed methodology is based on partial coverage of almost-simple paths of $\mathcal{A}_A$. In other words finding the most suitable path in $\mathcal{A}_{R \vee \bar{F} \vee \varphi}$ for every almost-simple path in $\mathcal{A}_A$, where φ is a sensible simple LTL formula that is proposed as a candidate for completion. Finally, the suitability will be assessed as the average of partial coverage over all edges on the path.

The output of such a procedure will be a sequence of candidate formulae, each associated with an estimated coverage (a number between 0 and 1) the addition of this particular formula would entail. The candidate formulae are added in a sequence so that the best unused formula is selected in every round. Finally, the coverage is always related to $\mathcal{A}_A$ and, thus, if some behaviour that cannot be observed in $\mathcal{A}_A$ is added with a candidate formula this addition will neither improve nor degrade the coverage.

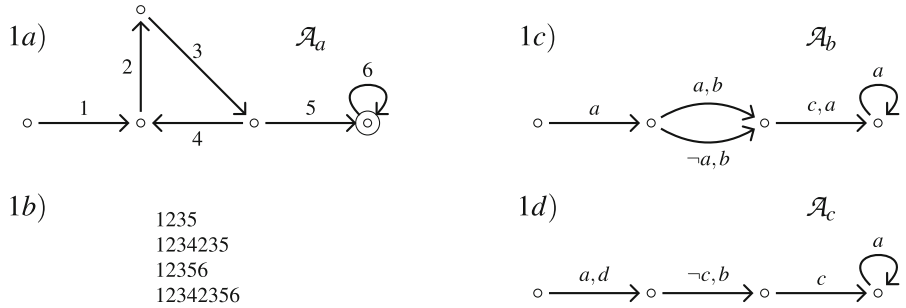

Fig. 2. **a** Example Büchi automaton $\mathcal{A}_a$; **b** all almost-simple paths of $\mathcal{A}_a$; **c**, **d** are two different Büchi automata with relatively similar evaluations of almost-simple paths (see Example 6)

Example 6 The method of Büchi automata evaluation will be partially exemplified using Fig. 2. The example only shows the enumeration of almost-simple paths and the partial coverage of two paths. What remains to the complete methodology will be shown more structurally in Algorithm 5. The enumeration of almost-simple paths of $\mathcal{A}_a$ in Fig.1a) should be straightforward, part the fact that a path is represented as a sequence of edges for simplicity. Let us assume that $\mathcal{A}_b$ is the original automaton and $\mathcal{A}_c$ is being evaluated for how thoroughly it covers $\mathcal{A}_b$. There are 4 almost-simple paths in $\mathcal{A}_b$, one of them is $\pi = \langle a;\ \neg a,\ b;\ \ c,\ a;\ \ a\rangle$. The partial coverage between the first edge of π and the first edge of $\mathcal{A}_c$ (there is only one possibility) is 0.5, since there is the excessive literal d. The coverage between the second edges is also 0.5, but only because of $\neg c$ in $\mathcal{A}_c$; the superfluous literal $\neg a$ restricts only the behaviour of $\mathcal{A}_b$. Finally, the average similarity between π and the respective path in $\mathcal{A}_c$ is 0.75 and it is approximately 0.7 between the two automata. △

The topmost level of the completeness evaluation methodology is shown as Algorithm 5. As input this function requires the three sets of user defined requirements, the set of candidate formulae and the number of formulae the algorithm needs to select. On lines 1 and 2 the formulae for conjunction of assumptions and user requirements (both required and forbidden) are created. They will be used later to form larger formulae to be translated into Büchi automata and evaluated for completeness but, for now, they need to be kept separate. Next step is to enumerate the almost-simple paths of $\mathcal{A}_A$ for later comparison, i.e. a baseline state space that the formulae from Γ_{Cand} should cover.

Algorithm 5: Completeness Evaluation

Input : $\Gamma_A, \Gamma_R, \Gamma_F, \Gamma_{Cand}, n$
Output: Best coverage for $1 \ldots n$ formulae from Γ_{Cand}

1 $\gamma_{Assum} \leftarrow \bigwedge_{\gamma \in \Gamma_A} \gamma$
2 $\gamma_{Desc} \leftarrow \bigwedge_{\gamma \in \Gamma_R} \gamma \vee \bigvee_{\gamma \in \Gamma_F} \gamma$
3 A $\leftarrow$ `transform2BA`(γ_{Assum})
4 pathsBA $\leftarrow$ `enumeratePaths`(A)
5 **for** $i = 1 \ldots n$ **do**
6 max $\leftarrow \infty$
7 **foreach** $\gamma \in \Gamma_{Cand}$ **do**
8 $\gamma_{Test} \leftarrow \gamma_{Desc} \vee \gamma$
9 A $\leftarrow$ `transform2BA`(γ_{Test})
10 cur $\leftarrow$ `avrPathCov`(A, pathsBA)
11 **if** max $<$ cur **then**
12 max $\leftarrow$ cur
13 $\gamma_{Max} \leftarrow \gamma$
14 `print`("Best coverage in i-th round is max.")
15 $\gamma_{Desc} \leftarrow \gamma_{Desc} \vee \gamma_{Max}$
16 $\Gamma_{Cand} \leftarrow \Gamma Cand \setminus \{\gamma_{Max}\}$

The rest of the algorithm forms a cycle that iteratively evaluates all candidates from Γ_{Cand} (see line 8 where the corresponding formula is being formed). Among the candidate formulae the one with the best coverage of the paths is selected and subsequently added to the covering system.

Functions `enumeratePaths` and `avrPathCov` are similar extensions of the BFS algorithms. Unlike BFS, however, they do not keep the set of visited vertices to allow state revisiting (twice in case of `enumeratePaths` and arbitrary number of times in case of `avrPathCov`). The `avrPathCov` search is executed once for every path it receives as input and stops after inspecting all paths to the length of the input path or if the current search path is incompatible (see Definition 8).

5. Experimental evaluation

All three sanity checking algorithms were implemented as an extension of the parallel explicit-state LTL model checker DiVinE [BBČR10]. From the many facilities offered by this tool, only the LTL to Büchi translation was used. Similarly as the original tool also this extension to sanity checking was implemented using parallel computation.

5.1. Experiments with random formulae

The first set of experiments was conducted on randomly generated LTL formulae. The motivation behind using random formulae is to demonstrate the reduction in the number of consistency checks. A naive algorithm for detecting inconsistent/redundant subsets requires an exponential number of such checks. Generating a large number of requirement collections with varying relations among individual requirements allows a more representative results than a few, real-world collections.

In order for the experiments to be as realistic as possible (avoiding trivial or exceedingly complex formulae) requirements with various nesting depths were generated. Nesting depth denotes the depth of the syntactic tree of a formula. Statistics about the most common formulae show, e.g. in [DAC98], that the nesting is rarely higher than 5 and is 3 on average. Following these observations, the generating algorithm takes as input the desired number n of formulae and produces: $n/10$ formulae of nesting 5, $9n/60$ of nesting 1, $n/6$ of nesting 4, $n/4$ of nesting 2 and $n/3$ of nesting 3. Finally, the number of atomic propositions is also chosen according to n (it is $n/3$) so that the formulae would all contribute to the same state space.

All experiments were run on a dedicated Linux workstation with quad core Intel Xeon 5130 @ 2GHz and 16GB RAM. The codes were compiled with optimisation options `-O2` using GCC version 4.3.2. Since the running times and even the number of checks needed for completion of all proposed algorithms differ for every set of formulae, the experiments were ran multiple times. The sensible number of formulae starts at 8: for less formulae the running time is negligible. Experimental tests for consistency and redundancy were executed for up to 15 formulae and for each number the experiment was repeated 25 times.

Figure 3 summarises the running times for consistency checking experiments. For every set of experiments (on the same number of formulae) there is one box capturing median, extremes and the quartiles for that set of experiments. From the figure it is clear that despite the optimisation techniques employed in the algorithm both median and maximal running times increase exponentially with the number of formulae. On the other hand there are some cases for which presented optimisations prevented the exponential blow-up as is observable from the minimal running times.

Figure 4 illustrates the discrepancy between the number of combinations of formulae and the number of redundancy checks that were actually performed. The number of combinations for n formulae is $n * 2^{n-1}$ but the optimisation often led to much smaller number. As one can see from the experiments on 9 formulae, it is potentially necessary to check almost all the combinations but the method proposed in this paper requires on average less than 10 % of the checks and the relative number decreases with the number of formulae.

The actual running times, as reported in Fig. 3, depend crucially on the chosen method for consistency checking: automata-based Algorithm 4 in our case. Our algorithm for enumerating all smallest inconsistent subsets is robust with respect to the consistency checking algorithm used. Hence any other consistency (or realisability) tool could be used, but that would only affect the actual running times, but not the number of checks, reported in Fig. 4.

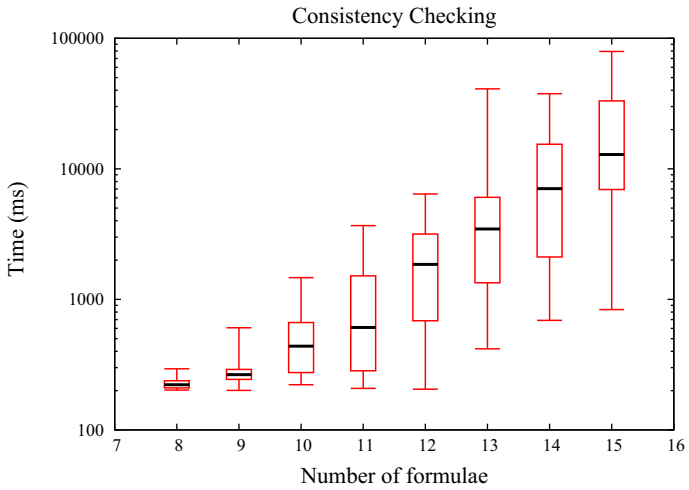

Fig. 3. Log-plot summarising the time complexity of consistency checking

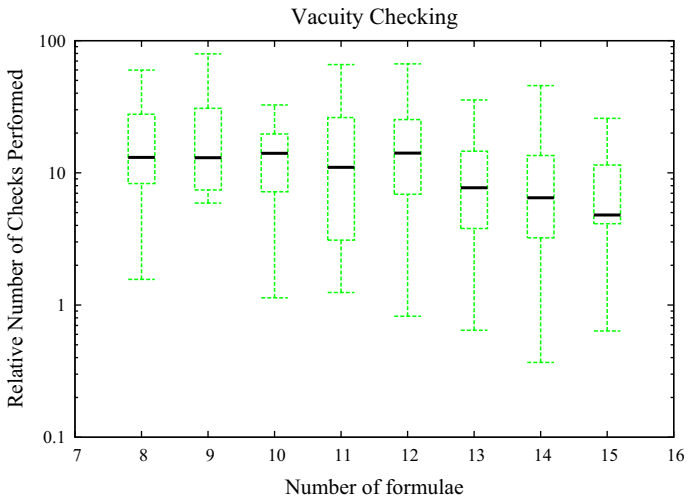

Fig. 4. Log-plot with the relative number of checks for redundancy checking

5.2. Case study: aeroplane control system

In order to demonstrate the full potential of the proposed sanity checking, we apply all the steps in a sequence. First, the whole requirements document is checked for consistency and redundancy and only the consistent and non-redundant subset is then evaluated with respect to its completeness. A sensible exposition of the effectiveness of completeness evaluation proposed in this paper requires more elaborate approach than using random formulae as the input requirements document.

For that purpose a case study has been devised that demonstrates the capacity to assess coverage of requirements and to recommend suitable coverage-improving LTL formulae. The requirements document was written by a requirements engineer whose task was to evaluate to method, and we only use random formulae as candidates for coverage improvement. The candidate formulae were built based on the atomic proposition that appeared in the input requirements and only very simple generated formulae were selected. It was also required that the candidates do not form an inconsistent or tautological set. Alternatively, pattern-based [DAC98] formulae could be used as candidates. The methodology is general enough to allow semi-random candidates generated using patterns from input formulae. For example if an implication is used as the input formula, the antecedent may not be required by the other formulae which may not be according to user's expectations.

Atomic Propositions

$a \equiv [\text{height} = 0]$
$b \equiv [\text{speed} \leq 200]$
$l \equiv [\text{landing}]$
$u \equiv [\text{undercarriage}]$
$c \equiv [\text{speed} \leq 100]$

LTL Requirements

$A1 : \mathcal{G}(a \Leftrightarrow b)$
$A2 : \mathcal{F}\, l \wedge \mathcal{G}(l \Rightarrow \mathcal{F}\, a)$
$A3 : \mathcal{G}(\neg l \Rightarrow \neg b)$
$A4 : \mathcal{G}(u \Rightarrow \mathcal{F}\, a \wedge u \Rightarrow c)$
$A5' : \mathcal{G}(\neg l \Rightarrow \neg a)$
$A6' : \mathcal{G}(\neg b \vee \mathcal{F}\ a)$

$R1 : \mathcal{G}(l \Rightarrow \mathcal{F}(\mathcal{G}\, b \wedge \mathcal{F}\, \mathcal{G}\, c))$
$R2 : \mathcal{G}(l \Rightarrow \mathcal{F}(b\, \mathcal{U} c))$
$R3 : \mathcal{G}(\neg b \Rightarrow \neg u)$
$R4 : \mathcal{F}(l\, \mathcal{U}(u\, \mathcal{U} c))$
$R5' : \mathcal{F}(l \wedge \mathcal{G}(\neg c \vee \mathcal{F}(\neg b)))$
$R6' : \mathcal{F}(\neg b \wedge \neg u \wedge \mathcal{X}(\neg b \wedge u) \wedge \mathcal{X}\,\mathcal{X}(b \wedge u))$

$F1 : \mathcal{F}(a \wedge \mathcal{G}(\neg a))$

Fig. 5. The two tables explain the shorthands for atomic propositions and list the LTL requirements

The case study attempts to propose a system of LTL formulae that should control the flight and more specifically the landing of an aeroplane. The LTL formulae and the atomic propositions they use are summarised in Fig. 5. The requirements are divided into 3 categories similarly as in the text: A requirements represent assumptions and R and F stand for required and forbidden behaviour, respectively. The grayed formulae were found either redundant or inconsistent and were not included in the completeness evaluation.

A1 The plane is on the ground only if its speed is less or equal to 200 mph.
A2 The plane will eventually land and every landing will result in the plane being on the ground.
A3 If the plane is not landing its speed is above 200 mph.
A4 Whenever the undercarriage is down, first the speed must be at most 100 mph and second the plane must eventually land on the ground.
A5 Whenever the plane is not landing its height above ground is not zero.
A6 At any time during the flight it holds that either the plane if flying faster than 200 mph or it eventually touches the ground.
R1 The landing process entails that eventually: (1) the speed will remain below 200 mph indefinitely and (2) after some finite time the speed will also remain below 100 mph.
R2 The landing process also entails that the plane should eventually slow down from 200 mph to 100 mph (during which the speed never goes above 200 mph).
R3 Whenever flying faster than 200 mph, the undercarriage must be retracted.
R4 At some point in the future, the plane should commence landing, then detract the undercarriage until finally the speed is below 100 mph.
R5 In the future the plane will be decommissioned: it will land and then either never flying faster than 100 mph or always slowing down below 200 mph.
R6 At some point in the future, the speed will be below 200 mph and the undercarriage retracted. Immediately before that and while the speed is still above 200 mph the undercarriage is detracted in two consecutive steps.
F1 The plane will eventually be on the ground after which it will never take off again.

Consistency The three categories of requirements are checked for consistency and redundancy separately. The environmental assumptions form a consistent set. The set of required behaviour contained two smallest inconsistent subsets. The requirement R4 is inconsistent with R6, hence we have choose R4 as more important and exclude R6 from the collection. The second inconsistent subset was: R1, R2, R5. There we chose to preserve R1 and R2, leaving out the possibility to decommission a plane.

Redundancy After regaining consistency of the required behaviour, our sanity checking procedure detects no redundancy in this set. On the other hand there are two redundant environmental assumptions. The assumption A5 is implied by the conjunction of A1 and A3 and is thus redundant. It also holds that any environment in which A2 and A3 are true also satisfies A6. Hence both A5 and A6 can be removed without changing the set of sensible behaviour in the environment.

Table 1. Iterations of the completeness evaluation algorithm for the aeroplane control system

Iteration	Suggested formula	Resulting system	Coverage of $\mathcal{A}_A$ by $\mathcal{A}_n$
1	$\varphi_1 = \mathcal{G}(\neg l)$	$\mathcal{A}_1 = \mathcal{A}_{R \vee \overline{F} \vee \overline{\varphi_1}}$	9.1 %
2	$\varphi_2 = \mathcal{F}(\neg b \wedge \neg l)$	$\mathcal{A}_2 = \mathcal{A}_{R \vee \overline{F} \vee \overline{\varphi_1} \vee \varphi_2}$	39.4 %
3	$\varphi_3 = \mathcal{F}(\neg l\, \mathcal{U}(a \vee b))$	$\mathcal{A}_3 = \mathcal{A}_{R \vee \overline{F} \vee \overline{\varphi_1} \vee \varphi_2 \vee \varphi_3}$	54.9 %

Completeness Initially, the remaining required and forbidden behaviour together does not cover the environment assumptions at all, i.e. the coverage is 0. There simply are so many possible paths allowed by the assumptions which are not considered in the requirements, that our metric evaluated the coverage to 0. Given the way the coverage is calculated, it must be the case that for every path in the assumption automaton there was not a single path in the requirements automaton that would not violate some of the edge labels of the assumptions. There is thus a considerable space for improvement. The completeness improving process we are about to describe begins by generating a set of candidate requirements among which those with the best resulting coverage are selected. Table 1 summarises the process.

The first formula selected by the Algorithm 5 and leading to coverage of 9.1 % was a simple $\varphi_1 = \mathcal{G}(\neg l)$: "The plane will never land". Not particularly interesting per se, nonetheless emphasising the fact that without this requirement, landing was never required. The formula φ_1 should thus be included among the forbidden behaviour. Unlike φ_1, the formula generated as second, $\mathcal{F}(\neg b \wedge \neg l)$: "At some point the plane will fly faster than 200 mph and will not be in the process of landing", should be included among the required behaviour. This formula points out that the required behaviour only specifies what should happen after landing, unlike the assumptions, which also require flight. The third and final formula was $\mathcal{F}(\neg l\, \mathcal{U}(a \vee b))$: "Eventually it will hold that the landing does not commence until either: (1) the plane is already on the ground or (2) the speed decreased below 200 mph". This last formula connects flight and landing and its addition (among the required behaviour) entails coverage of 54.9 %.

There are two conclusion one can draw from this case study with respect to the completeness part of sanity checking. First, the requirements are generated automatically, they are relevant and already formalised, thus amenable to further testing and verification. Second, the new requirements can improve the insight of the engineer into the problem. Both of these properties are valuable on their own, even though the method did not finish with 100 % coverage. The method produced further, coverage-improving formulae but even the best formulae only negligible improved the coverage and thus the engineer decided to stop the process.

5.3. Alternative LTL to Büchi translations

Although the running time experiments were one of the priorities of the original paper [BBB12a], their purpose was to investigate the effects of the proposed optimisations and not the concrete time measurements. In other words, the subject of our investigation was the asymptotic behaviour of the algorithms and how effectively does the algorithm scale with the number of formulae in the average and in the extrema. Given our recent cooperation with Honeywell and their interest in checking sanity of real-world, industry-level sets of requirements, the actual running times became of particular interest as well.

Initial experiments revealed, however, that on larger sets of requirements and especially when more complex individual requirements were involved, the proposed sanity checking algorithms were unable to cope with the computational complexity. The bottleneck proved to be the algorithm translating LTL formulae—conjunctions of the original requirements—into Büchi automata. The translator incorporated in DiVinE is sufficiently fast on small formulae that commonly occur in software verification and was never intended to be used with larger conjunctions. Thus, to enable checking sanity of real-world sets of requirements, we have reimplemented the algorithms to employ the state-of-the-art LTL to Büchi translator SPOT [DL11] in a tool called Looney.[2] As the case study summarising our cooperation with Honeywell in Sect. 5.4 shows, Looney is able to process much larger formulae than the original tool and also to incorporate a larger number of formulae into the sanity checking process.

The reimplementation of consistency was relatively straightforward since SPOT can be interfaced via a shared library with an arbitrary C++ application. Even though this enforced us to use the internal implementation of formulae from SPOT, which was different the one we had before, SPOT offers an adequate set of formulae-manipulating operations that allowed the translation of our algorithms, without their needing to be considerably

[2] As we are checking the sanity of requirements, the translator helps us to *spot* the *looney* ones. The tool is available at http://anna.fi.muni.cz/~xbauch/code.html#sanity.

modified. The reimplementation of the coverage algorithms was complicated by the fact that the product of SPOT is a *transition-based* Büchi automaton, whereas the algorithm of Sect. 4 expects *state-based* automata: the difference is that transitions rather than states are denoted as accepting.

This shift requires modification of the coverage evaluation, more precisely the definition of $\Lambda(\mathcal{A}_1, \mathcal{A}_2)$ of the coverage among automata. Before, the number m in the denominator referred to the number of almost-simple paths of $\mathcal{A}_1$ that ended in an accepting vertex (state). We propose to redefine m as the number of almost-simple paths that end in an accepting edge (transition). It is clear that the coverage reported by the original and the new metrics would differ: first, because the automata are generated using different methods and may thus be structurally different and, second, because the redefined metric considers different sets of almost-simple paths. With the alternative definition of m one can easily incorporate translators that produce transition-based automata (though parsing these automata may still require some degree of modification due to incompatibility of internal structures representing the automata).

Therefore, in order to reestablish admissibility of the new evaluation as an adequate coverage metric, we have repeated the experiment with the aeroplane control system. As was to be expected given the above argumentation, the concrete evaluation of individual formulae differed from our original experiments. Specifically, the formula first generated ($\mathcal{G}(\neg l)$) was evaluated with coverage 12.9 % (instead of 9.1); the second formula $\mathcal{F}(\neg b \wedge \neg l)$ with 35.0 (instead of 39.4); and the third formula $\mathcal{F}(\neg l\, \mathcal{U}(a \vee b))$ with 61.1 (instead of 54.9). Hence there are two observation to be made. First, even though the coverage numbers differ for individual formulae, the same triple was produced by the modified algorithm. Moreover, the three formulae were produced in the same order, thus the new metric behaves appropriately even when a new set of requirements is obtained as a disjunction of the old set with the generated coverage-improving formula.

5.4. Industrial evaluation

As a further extension to the original paper [BBB12a], we have evaluated our consistency and redundancy checking method on a real-life sets of requirements obtained in cooperation with Honeywell International. We did not extend this part of evaluation to the completeness checking since that would require writing a new set of requirements with separating the assumptions. The case study consisted of four Simulink models; each has been associated with a set of formalised requirements in the form of LTL formulae. The precise requirements are unfortunately confidential, but we provide statistics regarding the complexity of the requirement documents. The formalisation of the requirements that were originally given in natural language form has been done with the help of the ForReq tool. The tool allows the user to write formal requirements with the help of the specification patterns [DAC98] and some of their extensions. See [BBB+12b] for a detailed description of the tool.

Out of the four models, one (Lift) was a toy model used in Honeywell to evaluate the capabilities of ForReq, three (VoterCore, AGS, pbit) were models of real-world avionics subsystems. As the focus of this paper is model-less sanity checking, we ignored the models themselves and only considered the formalised requirements. The results of our experiments are summarised in Table 2.

The table reports that the largest collection of requirements consisted of 10 formulae, which can hardly be considered as a demonstration of scaling to industrial level. Even though some of the requirements were extremely complicated (with as many as 100 LTL operators), there is clearly a large space for improvement. It would certainly be possible to use a different algorithm for performing a single consistency check, but the number of these checks that needs to be performed is an independent issue left for future work. In all four cases, the whole set of requirements was consistent. As for redundancy, we have identified two cases where a formula was implied by another one. This is shown in the redundancy result column. The number displayed in the mem(MB) column represent the maximal (peak) amount of memory used.

The experiments, with the exception of the last one, were run on a dedicated Linux workstation with Intel Xeon E5420 @ 2.5 GHz and 8 GB RAM. The last experiment (pbit) was run on a Linux server with Intel X7560 @ 2.27 GHz and 450 GB RAM. The last experiment was a consistency check only as the redundancy checking took too much time. The reason for such great consumption of time (and memory) is that the formulae in the pbit set of requirements are rather complex, some of them even using precise specification of time. For more details about the time extension of LTL and its translation to standard LTL, see [BBB+12b].

Table 2. Results of consistency and redundancy checking on an industrial case study

Model name	No. of formulae	Consistency			Redundancy		
		Result	Time (s)	Mem (MB)	Result	Time (s)	Mem (MB)
VoterCore	3	OK	0.1	21.6	OK	0.4	22.4
AGS	5	OK	0.1	23.5	$\varphi_4 \Rightarrow \varphi_2$	0.7	23.6
Lift	10	OK	73.6	58.5	$\varphi_1 \Rightarrow \varphi_5$	9460.5	1474.5
pbit	10	OK	210,879.0	22,649.0	–	–	–

6. Conclusion

This paper further expands the incorporation of formal methods into software development. Aiming specifically at the requirements stage we propose a novel approach to sanity checking of requirements formalised in LTL formulae. Our approach is comprehensive in scope integrating consistency, redundancy and completeness checking to allow the user (or a requirements engineer) to produce a high quality set of requirements easier. The expert knowledge of LTL is not a prerequisite for using the method, given the existence of automated translator from English. On the other hand, there are certain limitations with respect to the number of requirements checked concurrently, and larger collections may require manually dividing into smaller sets.

Especially in the earliest stages of the requirements elicitation, when the engineers have to work with highly incomplete, and often contradictory sets of requirements, could our sanity checking framework be of great use. Other realisability checking tools can also (and with better running times) be used on the nearly final set of requirements but our process uniquely target these crucial early states. First, finding all inconsistent subsets instead of merely one helps to more accurately pinpoint the source of inconsistency, which may not have been elicited yet. Second, once working with a consistent set, the requirements engineer gets further feedback from the redundancy analysis, with every minimal redundancy witness pointing to a potential lack of understanding of the system being designed. Finally, the semi-interactive coverage-improving generation of new requirements attempts to partly automate the process of making the set of requirements as detailed as to prevent later development stages from misinterpreting a less thoroughly covered aspect of the system. We demonstrate this potential application on case studies of avionics systems.

The above described addition to the standard elicitation process may seem intrusive, but the feedback we have gather from requirements engineers at Honeywell were mostly positive. Especially given the tool support for each individual steps of the process—pattern-based automatic translation from natural language requirements to LTL and the subsequent fully- and semi-automated techniques for sanity checking—a fast adoption of the techniques was observed. Many of the engineers reported a non-trivial initial investment to assume the basics of LTL, but in most cases the resulting overall savings in effort outweighed the initial adoption effort. After the first week, the overhead of tool-supported formalisation into LTL was less than 5 minutes per requirement, while the average time needed for corresponding stages of requirements elicitation decreased by 18 %.

We have also accumulated a considerable amount of experience from employing the sanity checking on real-world sets of requirements during our cooperation with Honeywell International. Of particular importance was the realisation that the time requirements of our original implementation effectively prevented the use of sanity checking on industrial scale. In order to ameliorate the poor timing results we have reimplemented the sanity checking algorithms using the state-of-the-art LTL to Büchi translator SPOT, which accelerated the process by several orders on magnitude on larger sets of formulae. Another important observation was that the concept of model-based vacuity is desirable to be translated into the model-free environment more directly than as redundancy. We thus extend the sanity checking process with the capacity to generate vacuity witnesses (using the formally described theory of path-quantified formulae). Finally, we summarise these and other observations in a case consisting of four sets of requirements gathered during the development of industry-scale systems.

One direction of future research is the pattern-based candidate selection mentioned above. Even though the selected candidates were relatively sensible in presented experiments, using random formulae can produce useless results. Finally, experimental evaluation on real-life requirements and subsequent incorporation into a toolchain facilitating model-based development are the long term goals of the presented research. This paper also lacks formal definition of *total coverage* (which the proposed partial coverage merely approximates). We intend to formulate an appropriate definition using uniform probability distribution: which would also allow to compute

total coverage without approximation and would not be biased by concrete LTL to BA translation. That solution, however, is not very practical since the underlying automata translation is doubly exponential.

Acknowledgements

The research leading to these results has received funding from the European UnionâŁ™s Seventh Framework Program (FP7/2007-2013) for CRYSTAL—Critical System Engineering Acceleration Joint Undertaking under Grant agreement No. 332830 and from specific national programs and/or funding authorities.

References

[ALW89] Abadi M, Lamport L, Wolper P (1989) Realizable and unrealizable specifications of reactive systems. In: Proceedings of ICALP, pp 1–17

[BB09] Bormann J, Busch H (2009) Method for the determination of the quality of a set of properties, usable for the verification and specification of circuits. U. S. Patent No. 7,571,398 B2

[BBB12a] Barnat J, Bauch P, Brim L (2012) Checking sanity of software requirements. In: Proceedings of SEFM, pp 48–52

[BBB+12b] Barnat J, Beran J, Brim L, Kratochvíla T, Ročkai P (2012) Tool chain to support automated formal verification of avionics simulink designs. In: Proceedings of FMICS, pp 78–92

[BBČR10] Barnat J, Brim L, Češka M, Ročkai P (2010) DiVinE: parallel distributed model checker. In: Proceedings of HiBi/PDMC, pp 4–7

[BBDER01] Beer I, Ben-David S, Eisner C, Rodeh Y (2001) Efficient detection of vacuity in temporal model checking. Form. Methods Syst. Des 18(2):141–163

[BCG+10] Bloem R, Cimatti A, Greimel K, Hofferek G, Könighofer R, Roveri M, Schuppan V, Seeber R (2010) RATSY—a new requirements analysis tool with synthesis. In: Proceedings of CAV, pp 425–429

[BFG+01] Blom S, Fokkink W, Groote J, van Langevelde I, Lisser B, van de Pol J (2001) μCRL: a toolset for analysing algebraic specifications. In: CAV, vol 2102 of LNCS. Springer, New York, pp 250–254

[CAB+98] Chan W, Anderson RJ, Bea P, Burns S, Modugno F, Notkin D, Reese JD (1989) Model checking large software specifications. IEEE Trans. Softw Eng 24:498–520

[CCG+02] Cimatti A, Clarke EM, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) NuSMV 2: an opensource tool for symbolic model checking. In: CAV, vol 2404 of LNCS. Springer, New York, pp 241–268

[CKKV01] Chockler H, Kupferman O, Kurshan R, Vardi MY (2001) A practical approach to coverage in model checking. In: CAV, vol 2102 of LNCS. Springer, New York, pp 66–78

[CKV01] Chockler H, Kupferman O, Vardi MY (2001) Coverage metrics for temporal logic model checking. In: TACAS, vol 2031 of LNCS. Springer, New York, pp 528–542

[CRST08] Cimatti A, Roveri M, Schuppan V, Tchaltsev A (2008) Diagnostic information for realizability. In: Proceedings of VMCAI, pp 52–67

[CVWY92] Courcoubetis C, Vardi MY, Wolper P, Yannakakis M (1992) Memory-efficient algorithms for the verification of temporal properties. Form. Method Syst. Des 1:275–288

[DAC98] Dwyer MB, Avrunin GS, Corbett JC (1998) Property specification patterns for finite-state verification. In: Proceedings of FMSP, pp 7–15

[DL11] Duret-Lutz A (2011) LTL translation improvements in spot. In: Proceedings of VECoS, pp 72–83

[FG03] Feierbach G, Gupta V (2003) True coverage: a goal of verification. In: Proceedings of ISQED, pp 75–78

[HJC+08] Hinchey M, Jackson M, Cousot P, Cook B, Bowen JP, Margaria T (2008) Software engineering and formal methods. Commun. ACM 51:54–59

[HL95] Heimdahl MPE, Leveson NG (1995) Completeness and consistency analysis of state-based requirements. In: Proceedings of ICSE, pp 3–14

[KGG99] Katz S, Grumberg O, Geist D (1999) "Have I Written Enough Properties?"—a method of comparison between specification and implementation. In: Proceedings of CHARME, pp 280–297

[KHB09] Konighofer R, Hofferek G, Bloem R (2009) Debugging formal specifications using simple counterstrategies. In: Proceedings of FMCAD, pp 152–159

[Kup06] Kupferman O (2006) Sanity checks in formal verification. In: CONCUR, vol 4137 of LNCS. Springer, New York, pp 37–51

[KV03] Kupferman O, Vardi MY (2003) Vacuity detection in temporal model checking. STTT 4:224–233

[Lev00] Leveson N (2000) Completeness in formal specification language design for process-control systems. In: Proceedings of FMSP, pp 75–87

[LMS04] Lynce I, Marques-Silva JP (2004) On computing minimum unsatisfiable cores. In: Proceedings of SAT, pp 305–310

[LS08] Liffiton M, Sakallah K (2008) Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reasoning 40(1):1–33

[MTH03] Miller SP, Tribble AC, Heimdahl MPE (2003) Proving the shalls. In: FME, vol 2805 of LNCS. Springer, New York, pp 75–93

[RDB+05] Roy S, Das S, Basu P, Dasgupta P, Chakrabarti PP (2005) SAT based solutions for consistency problems in formal property specifications for open systems. In: Proceedings of ICCAD, pp 885–888

[RLS+03] Regimbal S, Lemire J-F, Savaria Y, Bois G, Aboulhamid E, Baron A (2003) Automating functional coverage analysis based on an executable specification. In: Proceedings of IWSOC, pp 228–234

[RV07] Rozier K, Vardi MY (2007) LTL satisfiability checking. In: SPIN, vol 4595 of LNCS. Springer, New York, pp 149–167

[RWH07] Rajan A, Whalen MW, Heimdahl MPE (2007) Model validation using automatically generated requirements-based tests. In: Proceedings of HASE, pp 95–104
[Sch12] Schuppan V (2012) Towards a notion of unsatisfiable and unrealizable cores for LTL. Sci. Comput. Program 77(7–8):908–939
[TK01] Tasiran S, Keutzer K (2001) Coverage metrics for functional validation of hardware designs. IEEE Des. Test. Comput 18(4):36–45
[WRHM06] Whalen MW, Rajan A, Heimdahl MPE, Miller SP (2006) Coverage metrics for requirements-based testing. In: Proceedings of ISSTA, pp 25–36

Received 29 August 2013
Accepted in revised form 3 November 2015 by George Eleftherakis, Mike Hinchey, and Michael Butler
Published online 4 January 2016