# Checking Sanity of Software Requirements*

Jiří Barnat, Petr Bauch, Luboš Brim

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{barnat,xbauch,brim}@fi.muni.cz

**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 techniques that automatically detect flaws and suggest improvements of given requirements. Specifically, we describe and experimentally evaluate new approaches to consistency and vacuity checking that identify all inconsistencies and pinpoint their exact source (the smallest inconsistent set). To complete the sanity checking we also deliver a novel 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.

## 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 [10].

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 [1, 6] or theorem proving [2] 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.

* This work has been supported by the Czech Grant Agency grant No. GAP202/11/0312 and GD102/09/HD042 and by Artemis-IA iFEST project grant No. 100203.

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. To mitigate the situation a subsidiary approach, the sanity checking, was proposed to check vacuity and coverage of requirements [12]. 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 design, implement and evaluate techniques that would allow the developers of computer systems to check sanity of their requirements when it matters most, i.e. during the requirements stage. The implementation of our techniques is planned to be incorporated in the iFEST project [11]. Together with a tool translating natural language requirements into LTL formulae our techniques could provide automatic sanity checking procedure of freshly elicited requirements.

*Contribution* This paper redefines the notion of sanity checking of requirements written as LTL formulae and describes its implementation and evaluation. The proposed notion liberates sanity checking from the necessity of having a model of the developed system. Sanity checking commonly consists of three parts: consistency and vacuity checking and completeness of requirements. Our approach to consistency and vacuity checking is novel in identifying all inconsistent (or vacuous) subsets of the input set of requirements. This considerably simplifies the work of requirements engineers because it pinpoints all sources of inconsistencies. For completeness checking, we propose a new behaviour-based coverage metric. Assuming that the user specifies what behaviour of the system is sensible, our coverage metric calculates what portion of this behaviour is described by the requirements specifying the system itself. The method further suggests new requirements to the user that would improve the coverage and thus ensure more accurate description of users' expectations. The efficiency and usability of our approach to sanity checking is verified in an experimental evaluation and a case study.

### 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 [3]. 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 [20, 16].

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 [4, 5]. Vacuity of temporal formulae was identified as a problem related to model checking and solutions were proposed in [13] and in [12], 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 [17] or in [18] 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.

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 [9, 14, 15]. 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.

## 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 this recursive definition specifies all well-formed* LTL formulae *over AP, where* $p \in AP$*:*

$$\Psi ::= p | \neg\Psi | \Psi \wedge \Psi | X\ \Psi | \Psi\ U\ \Psi$$

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

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* $\Sigma$ *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*

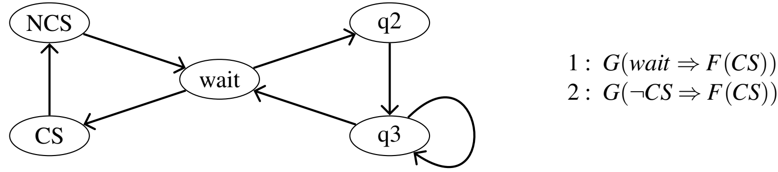


Fig. 1: LTS for Peterson's mutual exclusion protocol (with only one process) and two liveness LTL formulae.

*relation,* $\nu : S \to 2^{\Sigma}$ *is a valuation function and* $s_0 \in S$ *is the initial state. A function* $r : \mathbb{N} \to S$ *is an* infinite run *over the states of* $M$ *if* $r(0) = s_0, \forall i : r(i) \to r(i+1)$*. The* trace *or* word *of a run is a function* $w : \mathbb{N} \to 2^{\Sigma}$*, 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 the LTS is 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* $\Psi$ *be an LTL formula over* $\Sigma$*. Then it is possible to decide if* $w$ *satisfies* $\Psi$ *(*$w \models \Psi$*) based on the following rules (where* $w(i)$ *is the i-th letter of* $w$ *and* $w_i$ *is the i-th suffix of* $w$*):*

$$
\begin{array}{lll}
w \models p & \textit{iff} & p \in w(0), \\
w \models \neg\Psi & \textit{iff} & w \not\models \Psi, \\
w \models \Psi_1 \wedge \Psi_2 & \textit{iff} & w \models \Psi_1 \textit{ and } w \models \Psi_2, \\
w \models X\ \Psi & \textit{iff} & w_1 \models \Psi, \\
w \models \Psi_1\ U\ \Psi_2 & \textit{iff} & \exists i \forall j < i : w_j \models \Psi_1, w_i \models \Psi_2,
\end{array}
$$

*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, e.g. 1: if a process *waits* for CS then it will eventually get there.

Clearly, a system as a whole satisfies an LTL formula if all its executions (all infinite words over the states of its LTS) do. 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 $\phi$ can be transformed into a Büchi automaton $A_{\phi}$ such that $w \models \phi \Leftrightarrow w \in L(A_{\phi})$. Also checking that every execution satisfies $\phi$ is equivalent to

checking that no execution satisfies $\neg\phi$. It only remains to combine the LTS model of the given system $M$ with $A_{\neg\phi}$ in such a way that the resulting automaton will accept exactly those words of $M$ that violate $\phi$. 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.

### 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 only for comparison with their model-free versions and interested reader should consult for example [12] for more details.

Let $K$ be an LTS, $\varphi$ a formula and $\psi$ its subformula. Then $\psi$ *does not affect the truth value of* $\varphi$ *in* $K$ if $K$ satisfies $\forall x\varphi[\psi/x]$ if and only if $K$ satisfies $\varphi$, where $[\psi/x]$ substitutes $x$ with $\psi$. Then a system $K$ *satisfies a formula* $\varphi$ *vacuously* iff $K \models \varphi$ and there is a subformula $\psi$ of $\varphi$ such that $\psi$ does not affect $\varphi$ in $K$.

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

It can be observed that these notions of sanity are deeply dependent on the system that is being verified. In order to be used without a model these notions need to be altered considerably while preserving the main idea. Vacuity states that the satisfaction of a formula is given extrinsically and is not related to the formula itself. Coverage, on the other hand, attempts to capture the amount of system behaviour that is described by the formulae. These concepts are in this paper translated into model-free environment and supplemented with consistency verification to form a complex sanity checking.

## 3 Model-free Sanity Checking

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*, *vacuity* and *completeness* checking.

**Definition 5** *A set* $\Gamma$ *of LTL formulae over AP is* consistent *if* $\exists w \in AP^{\omega} : w$ *satisfies* $\bigwedge\Gamma$. *Checking consistency of a set* $\Gamma$ *entails finding all minimal inconsistent subsets of* $\Gamma$. *A*

*formula* $\phi$ *is* vacuous *with respect to a set of formulae* $\Gamma$ *if* $\bigwedge\Gamma \Rightarrow \phi$*. To check vacuity of a set* $\Gamma$ *entails finding all pairs of* $\langle \phi \in \Gamma, \Phi \subseteq \Gamma \rangle$ *such that* $\Phi$ *is consistent and* $\Phi \Rightarrow \phi$ *(and for no* $\Phi' \subseteq \Phi$ *does it hold that* $\Phi' \Rightarrow \phi$*).*

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 vacuity is especially efficient if a large set of requirements needs to be processed and standard sanity checking would only reveal if there is an inconsistency (or vacuity) 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 vacuous 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 $\{p,q,a\}$. They are $1 : F(p \Rightarrow p\ U\ q)$, $2 : GF(p)$, $3 : G\neg(a \wedge p)$, $4 : G(X\ q \Rightarrow a)$ and $5 : GF(q)$. In this set the formula 4 is inconsistent due to the first 3 formulae and the last formula is vacuously satisfied (implied) by the first 2 formulae.

### 3.1 Implementation of Sanity Checking

Let us henceforth denote one specific instance of consistency (or vacuity) checking as a *check*. For consistency and a set $\Gamma$ it means to check that for some $\gamma \subseteq \Gamma$ is $\bigwedge\gamma$ satisfiable. For vacuity it means for $\gamma \subseteq \Gamma$ and $\phi \in \Gamma$ to check that $\bigwedge\gamma \Rightarrow \phi$ is satisfiable. In the worst case both consistency and vacuity 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 vacuity 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 $\Gamma$) 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 initiallised with all single element subsets of $\Gamma$ and $\Gamma$ itself, thus in the subsequent iteration will be checked the supersets of the former and subsets of the latter.

**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)
```

**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=$\langle i_1,\dots,i_j\rangle$)

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

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 3 of Algorithm 3).

The actual consistency (and quite similarly also vacuity) checking is less complicated (see Algorithm 4). First, the conjunction of formulae encoded in the task is created, then the appropriate Büchi automaton is checked for existence of an accepting cycle: using nested DFS [7]. The only difference when performing the vacuity checking is that the task t consists of a list $\langle i_1,\dots,i_j\rangle$ which can be empty, and one index $i_k$. Since the task is to check that $\phi_{i_1} \wedge \dots \wedge \phi_{i_j} \Rightarrow \phi_{i_k}$ the line 1 needs to be altered to $\mathsf{F} \leftarrow$ createConj$(\phi_{i_1},\dots,\phi_{i_j},\neg\phi_{i_k})$. Because if $\phi_{i_1} \wedge \dots \wedge \phi_{i_j} \wedge \neg\phi_{i_k}$ is satisfiable then $\phi_{i_1} \wedge \dots \wedge \phi_{i_j} \not\Rightarrow \phi_{i_k}$, i.e. $\phi_{i_k}$ is not satisfied vacuously by $\{\phi_{i_1},\dots,\phi_{i_j}\}$.

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 $+1$).

## 4 Completeness Checking

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 $\Gamma_A$, required behaviour $\Gamma_R$ and forbidden behaviour $\Gamma_F$. 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*". The required behaviour represents the functional requirements imposed on the system: the system will not be correct if any of these is not satisfied. Dually, the forbidden behaviour contains those patterns that the system must not display. 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. $\mathcal{A}_{A \vee \bar{B} \wedge \phi}$ accepts words satisfying $\bigwedge_{\gamma \in \Gamma_A} \gamma \vee \bigvee_{\gamma \in \Gamma_B} \gamma \wedge \phi$. 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 $\phi$ that $\mathcal{A}_{R \vee \bar{F} \vee \phi}$ 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 [5, 19] 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 6** *Let G be a directed graph. A* path $\pi$ *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 incorporate infinity. Clearly, it is a heuristic approach and a considerable amount of information will be lost but since all simple cycles will be considered the resulting evaluation should provide sufficient distinguishing capacity (as demonstrated in Section 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 (that will restrict the behaviour of the former). 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 evaluation compatible with the edge evaluation (now in the LTL automaton).

**Definition 7** *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 $\Lambda$ 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 (where $p = |A_1 \cap A_2|$):*

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

From the definition one can observe that $\Lambda$ 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 automata is based on averaging over almost-simple paths. An almost-simple path $\pi_2$ of automaton $\mathcal{A}_2$ covers an almost-simple path $\pi_1$ 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 $\pi_j$. 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 per cent 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 $\mathcal{A}_{R \vee \bar{F} \vee \phi}$ for every almost-simple path in $\mathcal{A}_A$, where $\phi$ 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

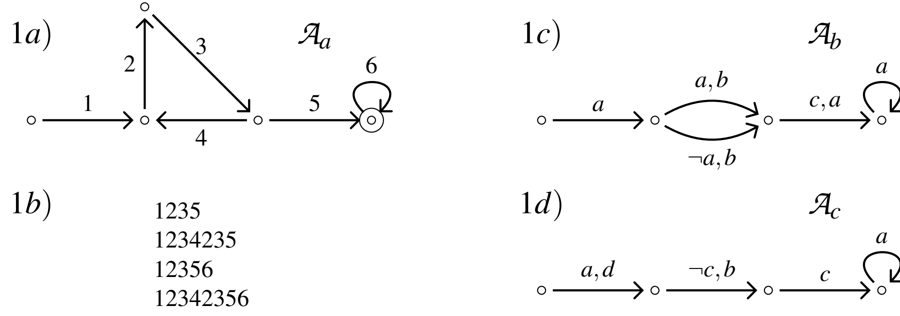


Fig. 2: $1a)$ Example Büchi automaton $\mathcal{A}_a$; $1b)$ All almost-simple paths of $\mathcal{A}_a$; $1c)$ and $1d)$ are two different Büchi automata with relatively similar evaluations of almost-simple paths (see Example 4).

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.

*Example 4.* The method of Büchi automata evaluation will be partially exemplified using Figure 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 Figure $1a)$ 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 $\pi$ 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 $\pi$ 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 $\Gamma_{Cand}$ should cover.

The rest of the algorithm forms a cycle that iteratively evaluates all candidates from $\Gamma_{Cand}$ (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

**Algorithm 5:** Completeness Evaluation

```
   Input   : Γ_A, Γ_R, Γ_F, Γ_Cand, n
   Output: Best coverage for 1...n formulae from Γ_Cand
1  γ_Assum ← ⋀_{γ∈Γ_A} γ
2  γ_Desc ← ⋀_{γ∈Γ_R} γ ∨ ⋁_{γ∈Γ_F} γ
3  A ← transform2BA(γ_Assum)
4  pathsBA ← enumeratePaths(A)
5  for i = 1...n do
6      max ← ∞
7      foreach γ ∈ Γ_Cand do
8          γ_Test ← γ_Desc ∨ γ
9          A ← transform2BA(γ_Test)
10         cur ← avrPathCov(A, pathsBA)
11         if max < cur then
12             max ← cur
13             γ_Max ← γ
14     print( "Best coverage in i-th round is max." )
15     γ_Desc ← γ_Desc ∨ γ_Max
16     Γ_Cand ← ΓCand \ {γ_Max}
```

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 7).

# 5 Experimental Evaluation

All three sanity checking algorithms were implemented as an extension of the parallel explicit-state LTL model checker DiVinE [1]. From the many facilities offered by this tool, only the LTL to Büchi translation was used. As the original tool also its extension was implemented using parallel computation, yet due to space limitation this aspect is not to be described in this paper.

## 5.1 Experiments with Random Formulae

The first set of experiments was conducted on randomly generated LTL formulae. In order for the experiments to be as realistic as possible formulae 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 [8], 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

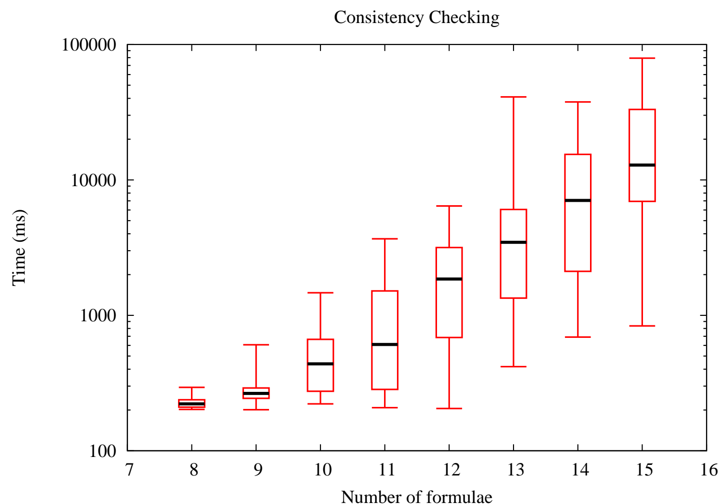


Fig. 3: Log-plot summarising the time complexity of consistency checking.

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 vacuity 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 vacuity 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 per cent of the checks and the relative number decreases with the number of formulae.

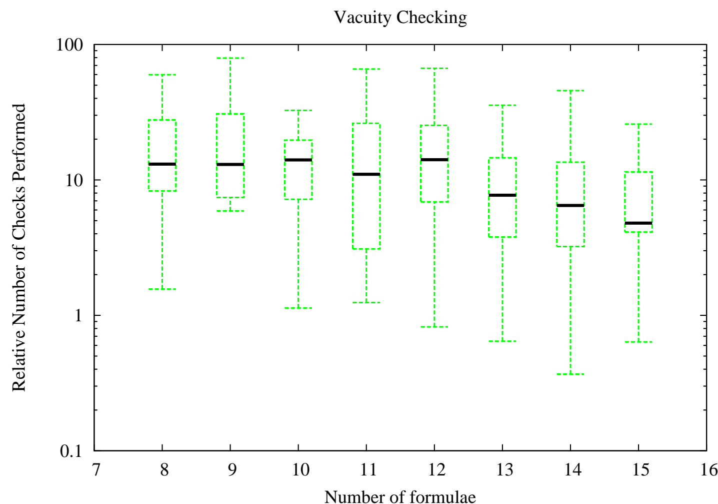


Fig. 4: Log-plot with the relative number of checks for vacuity checking.

### 5.2 Case Study: Aeroplane Control System

A sensible exposition of the effectivity of completeness evaluation proposed in this paper requires more elaborate approach than using random formulae. 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. Random formulae were used only as candidates for coverage improvement: they were built based on the atomic proposition that appeared in the input formulae 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 [8] 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.

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 Figure 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. For example the formula $R2$ expresses the requirement that whenever *landing* the plane should eventually slow down from 200 mph to 100 mph (during which the speed never goes above 200 mph).

Initially, the coverage of $R$ and $F$ requirements is 0. Though they were not selected specifically for the purpose of covering the $A$ requirements, it is still alarming that not a single path was preserved. The first formula selected by the Algorithm 5 and leading to coverage of 9.1 per cent was a simple $G(\neg l)$. Not particularly interesting per se, nonetheless emphasising the fact that without this requirement, landing was never required. Unlike the previous formula which would be added to forbidden behaviour,

| Atomic Propositions | LTL Requirements | |
|---|---|---|
| $a \equiv$ [height $= 0$] | $A1 : G(a \Leftrightarrow b)$ | $R1 : G(l \Rightarrow F(G(b) \wedge F(G(c))))$ |
| $b \equiv$ [speed $\leq 200$] | $A2 : F(l) \wedge G(l \Rightarrow F(a))$ | $R2 : G(l \Rightarrow F(b\ U\ c))$ |
| $l \equiv$ [landing] | $A3 : G(\neg l \Rightarrow \neg b)$ | $R3 : G(\neg b \Rightarrow \neg u)$ |
| $u \equiv$ [undercarriage] | $A4 : G(u \Rightarrow F(a) \wedge u \Rightarrow c)$ | $R4 : F(l\ U\ (u\ U\ c))$ |
| $c \equiv$ [speed $\leq 100$] | $F1 : F(a \wedge F(\neg a))$ | |

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

the next select formula $(F(\neg b \wedge \neg l))$ is clearly required totalling the coverage to 39.4 per cent. This formula points out that the required behaviour only specifies what should happen after landing, unlike assumption which also require flight. The final formula $F(\neg l\ U\ (a \vee b))$ connects flight and landing and its addition entails coverage of 54.9 per cent.

## 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, vacuity and completeness checking to allow the user (or a requirements engineer) to produce a high quality set of requirements easier. The novelty of our consistency (vacuity) checking is that they produce all inconsistent (vacuous) 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.

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.

## References


1. J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker. In *Proc. of HiBi/PDMC*, pages 4–7, 2010.

2. S. Blom, W. Fokkink, J. Groote, I. van Langevelde, B. Lisser, and J. van de Pol. $\mu$CRL: A Toolset for Analysing Algebraic Specifications. In *CAV*, volume 2102 of *LNCS*, pages 250–254. Springer, 2001.
3. W. Chan, R. J. Anderson, P. Bea, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model Checking Large Software Specifications. *IEEE T. Software Eng.*, 24:498–520, 1998.
4. H. Chockler, O. Kupferman, R. Kurshan, and M. Y. Vardi. A Practical Approach to Coverage in Model Checking. In *CAV*, volume 2102 of *LNCS*, pages 66–78. Springer, 2001.
5. H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage Metrics for Temporal Logic Model Checking. In *TACAS*, volume 2031 of *LNCS*, pages 528–542. Springer, 2001.
6. A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *CAV*, volume 2404 of *LNCS*, pages 241–268. Springer, 2002.
7. C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Form. Method Syst. Des.*, 1:275–288, 1992.
8. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-State Verification. In *Proc. of FMSP*, pages 7–15, 1998.
9. M. P. E. Heimdahl and N. G. Leveson. Completeness and Consistency Analysis of State-Based Requirements. In *Proc. of ICSE*, pages 3–14, 1995.
10. M. Hinchey, M. Jackson, P. Cousot, B. Cook, J. P. Bowen, and T. Margaria. Software Engineering and Formal Methods. *Commun. ACM*, 51:54–59, 2008.
11. industrial Framework for Embedded Systems Tools. http://www.artemis-ifest.eu.
12. O. Kupferman. Sanity Checks in Formal Verification. In *CONCUR*, volume 4137 of *LNCS*, pages 37–51. Springer, 2006.
13. O. Kupferman and M. Y. Vardi. Vacuity Detection in Temporal Model Checking. *STTT*, 4:224–233, 2003.
14. N. Leveson. Completeness in Formal Specification Language Design for Process-Control Systems. In *Proc. of FMSP*, pages 75–87, 2000.
15. S. P. Miller, A. C. Tribble, and M. P. E. Heimdahl. Proving the Shalls. In *FME*, volume 2805 of *LNCS*, pages 75–93. Springer, 2003.
16. A. Rajan, M. W. Whalen, and M. P. E. Heimdahl. Model Validation using Automatically Generated Requirements-Based Tests. In *Proc. of HASE*, pages 95–104, 2007.
17. S. Roy, S. Das, P. Basu, P. Dasgupta, and P. P. Chakrabarti. SAT Based Solutions for Consistency Problems in Formal Property Specifications for Open Systems. In *Proc. of ICCAD*, pages 885–888, 2005.
18. K. Rozier and M. Y. Vardi. LTL Satisfiability Checking. In *SPIN*, volume 4595 of *LNCS*, pages 149–167. Springer, 2007.
19. S. Tasiran and K. Keutzer. Coverage Metrics for Functional Validation of Hardware Designs. *IEEE Des. Test. Comput.*, 18(4):36–45, 2001.
20. M. W. Whalen, A. Rajan, M. P. E. Heimdahl, and S. P. Miller. Coverage Metrics for Requirements-Based Testing. In *Proc. of ISSTA*, pages 25–36, 2006.