Consistency Checking in Requirements Analysis

Jaroslav Bendík
Faculty of Informatics, Masaryk University
Brno, Czech Republic
xbendik@fi.muni.cz

ABSTRACT

In the last decade it became a common practise to formalise software requirements using a mathematical language of temporal logics, e.g., LTL. The formalisation removes ambiguity and improves understanding. Formal description also enables various model-based techniques, like formal verification. Moreover, we get the opportunity to check the requirements earlier, even before any system model is built. This so called requirements sanity checking aims to assure that a given set of requirements is consistent, i.e., that a product satisfying all the requirements can be developed. If inconsistencies are found, it is desirable to present them to the user in a minimal fashion, exposing the core problems among the requirements. Such cores are called minimal inconsistent subsets (MISes). In this work, we present a framework for online MISes enumeration in the domain of temporal logics.

CCS CONCEPTS

• **Software and its engineering** → **Requirements analysis**; *Formal software verification*;

KEYWORDS

requirements analysis; consistency; minimal inconsistent subsets

ACM Reference format:
Jaroslav Bendík. 2017. Consistency Checking in Requirements Analysis. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17-DOC),* 4 pages.
https://doi.org/10.1145/3092703.3098239

1 INTRODUCTION

Establishing requirements is an important stage in all development. The importance of a clear and precise specification is apparent from the necessity of a final-product compliance verification. Yet the specification itself is rarely described in an unambiguous way as a natural language is usually used to express the requirements. Still, the formal description is an essential for any kind of comprehrensive requirements verification [16]. Recent years have seen a tendencies to use the mathematical langauage of temporal logics, e.g., the *Linear Temporal Logic* [20] (LTL), to specify functional system requirements. Formal description enables various model-based techniques, such as model checking [13] or theorem proving [11]. Moreover, we also get the opportunity to check the requirements earlier, even before any system model is built. This so-called requirements sanity checking [3] aims to assure that a given set of requirements is consistent and that there are no redundancies. The presence of redundant requirements can possibly lead to further problems in future development and the inconsistency even means that no product satisfying all the requirements can be developed. If redundancies or inconsistencies are found, it is desirable to present them to the user in a concise fashion, exposing the core problem among the requirements. Such cores are called *minimal inconsistent subsets* (MISes) or also *minimal unsatisfiable subsets* of a given set of requirements. As redundancy checking can be usually reduced to inconsistency checking [2], the goal is thus to find all MISes.

We illustrate the inconsistency checking on an example. Assume that we are going to build a system that contains one peculiar component that has a very expensive initialisation phase and also a very expensive shutdown phase. We formalise the requirements using the branching temporal logic CTL [13]. In the formulae we use the atomic propositions q denoting that a *query* has arrived, r denoting that the component is *running*, and m denoting that the system is taken down for *maintenance*. Our first requirement states that whenever a query arrives, the component has to become active eventually, formally $\varphi_1 := \mathbf{AG}(q \rightarrow \mathbf{AF}\, r)$. The second requirement states that once the component is started, it may never be stopped. This may be a reasonable requirement, e.g.,if the component's initialisation is expensive, formally $\varphi_2 := \mathbf{AG}(r \rightarrow \mathbf{AG}\, r)$. The third requirement states that the system has to be taken down for maintenance once in a while. This also means that the component has to become inactive at that time. This is formalised as $\varphi_3 := \mathbf{AG}\,\mathbf{AF}\,(m \wedge \neg r)$. Our last requirement states that after the maintenance, the system (including the component we are interested in) has to be restarted, formally $\varphi_4 := \mathbf{AG}(m \rightarrow \mathbf{AF}\,(\neg m \wedge r))$. The situation is illustrated in Fig. 1. We discover that there is one minimal inconsistent subset of the four requirements, namely $\{\varphi_2, \varphi_3, \varphi_4\}$, and that there are three maximal consistent subsets of the requirements, namely $\{\varphi_1, \varphi_2, \varphi_3\}$, $\{\varphi_1, \varphi_2, \varphi_4\}$, $\{\varphi_1, \varphi_3, \varphi_4\}$. The consistency of the first set $\{\varphi_1, \varphi_2, \varphi_3\}$ might be surprising, as one would suspect the pair of requirements φ_2 and φ_3 to be the source of inconsistency. However, the first three requirements can hold at the same time – in systems where no queries arrive at all. In these situations we say that the requirements hold *vacuously*. There are ways of dealing with vacuity, such as employing the so-called vacuity witnesses [6].

There are two main issues concerning MISes enumeration. First, the complete enumeration of MISes is generally intractable due to the potentially exponentially many results. On the other hand, the more MISes are found the better insight into the inconsistency

ISSTA'17-DOC, July 2017, Santa Barbara, CA, USA

ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/10.1145/3092703.3098239

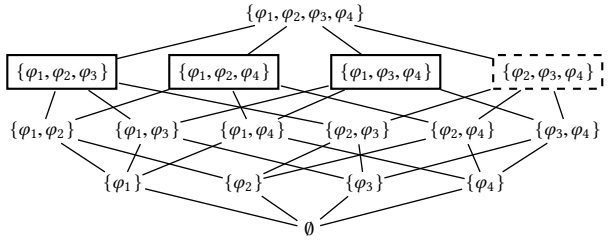

Figure 1: Illustration of the requirements analysis example. The subset with dashed outline is the minimal inconsistent one, the subsets with solid outline are the maximal consistent ones.

among requirements is provided. Therefore, it makes sense to study algorithms that output MISes in an online manner (i.e., incrementally). Second, every algorithm for finding MISes has to perform several consistency checks during its execution. In case of temporal logics, the consistency checks are usually very expensive to perform, especially compared to other domains like SAT or SMT. Therefore, a good algorithm for MISes enumeration should tend to minimise the number of performed consistency checks.

In this paper we focus on online enumeration of MISes of a given inconsistent set of requirements which are expressed in some temporal logic like LTL. Besides the attempt to enumerate as many MISes as possible we tend to minimise the number of performed consistency checks during the computation.

2 RELATED WORK

The problem of MISes finding is well known in the area of *constraint processing* where a satisfiability of a set of constraints is examined. Several algorithms for finding MISes of an overconstrained set have been developed. However, the majority of existing approaches were tailored for a specific type of mathematical language (constraints) which is not suitable for modeling of system requirements, e.g., SAT or SMT constraints. Also, many of these approaches focus only on extracting a single MIS of an overconstrained system [7, 19].

There are some algorithms for complete MIS enumeration that are applicable to any type of constraints, e.g., the MARCO [18] and the DAA [1] algorithms. These algorithms can be viewed more like a general schema that can employ as a subroutine any single MIS extraction algorithm, which makes them *domain agnostic*. However, we are not aware of any existing efficient single MIS extraction algorithm that is designed for temporal logic constraints. Also, these algorithms do not tend to minimise the number of performed consistency checks which is the most important criterion in the case of MISes enumeration of constraints expressed in a temporal logic.

Recently, a tool for online LTL MISes enumeration, called Looney, was presented [2]. However, Looney is built on an explicit enumeration of every subset of a given set of requirements which makes it suitable only for small instances as the number of subsets is exponential to the number of requirements. We made an experimental evaluation [8] of Looney where we also presented the basic version of our MISes enumeration algorithm called TOME; TOME has shown to be substantially faster than Looney.

Some projects, e.g., the one of Schuppan [22] or the one of Hantry and Hacid [15], focus on finding unsatisfiable and unrealizable cores of LTL formulas. However, they do not guarantee that a minimal core, i.e., a MIS, is returned.

There are several frameworks that focus on the formalization of software requirements. Such a framework usually accepts as an input a set of requirements expressed in some natural language-like format which has to, however, obey some very restricted grammar. This grammar restriction allows to subsequently automatically convert the requirements into a language of some temporal logic. Such functionality is, for example, provided by the ForReq framework [4]. Our goal is not to implement such framework; instead we would like to develop efficient algorithms for MISes enumeration that could be subsequently integrated into these frameworks.

Finally, there are several techniques that can be used for checking consistency of requirements expressed in a temporal logic. In case of LTL or CTL formulae, perhaps the oldest but also the most common approach is to reduce the consistency checking problem to a model checking problem[13] and employ one of several existing model checking tools (e.g., DIVINE [5], NuSMV [12] or SPOT [14]). Besides the reduction to the model checking problem, there have been recently proposed some completely different approches for consistency checking [10, 17, 21]. Any of these techniques can be employed in MISes enumeration and any future improvements in this area can bootstrap the MISes enumeration algorithms.

3 RESEARCH QUESTIONS AND CONTRIBUTIONS

In a current state-of-the-practise there exist tools that allow formalisation of software requirements in a semi-automatic way. There are also tools for checking the consistency of a given set of formalized requirements. However, there is a very limited support for MISes identification, i.e., the identification of the sources of the inconsistency of an inconsistent set of requirements in the domain of temporal logics. This work focuses on this gap and in particular, it aims to address the following research questions:

RQ1: What are the differences between temporal logics and logics for which efficient MIS enumeration algorithms already exist, and what techniques used in the latter setting can be applied also in the former one?

RQ2: Are there some domain specific properties of temporal logics, e.g., negation normal form of LTL formulae, that can be exploited by a singe MIS extraction algorithms?

RQ3: Which consistency checking approaches are the most suitable to be employed by MIS enumeration algorithms?

The main contributions of the research direction, both those that have been already achieved and those that are expected, are the following:

- We provide novel algorithms for online MISes enumeration which are applicable to any type of input logic. These algorithms are especially suitable for temporal logics, like LTL, as they tend to minimise the number of performed consistency checks during their execution which is the most relevant criterion in the case of temporal logics domain. We pioneered our work in [8] where an algorithm

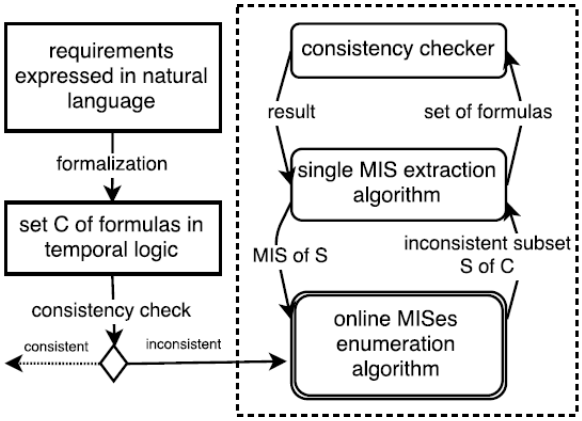

Figure 2: A high-level architecture of the MISes enumeration framework and its connection to the requirements analysis.

that outperforms the current state-of-the-art tool for LTL MISes enumeration was presented. Extended version of this algorithm, called TOME, was presented in [9]. TOME is able to employ any single MIS extraction procedure as its subroutine.

- We expect to provide efficient single MIS extraction algorithms in the domain of temporal logics. In particular, these algorithms should exploit specific properties of individual logics.
- We also plan to make a comprehensive experimental evaluation of LTL consistency checking techniques. Some comparisons of these techniques have been already done in other works [21, 23]. However, those comparisons target only the problem of checking a single set of requirements for consistency. On the other hand, the MIS finding problem naturally subsumes performing of a series of consistency checks, therefore a consistency checking approach with an incremental nature can be preferred to the best single-use approach.

4 METHODOLOGY AND EVALUATION

Our research is expected to result in a framework for online MISes enumeration of an inconsistent set of requirements. We use the term 'framework' to emphasize that it can be used with any type of requirements specification language as the core MISes enumeration feature will be a constraint agnostic algorithm, e.g., TOME.

4.1 Methodology

Fig. 2 shows the high-level architecture of the framework and its connection to the process of requirements analysis. In the following text we describe the several steps shown in Fig. 2 that form an automated framework for finding the sources of inconsistencies of a a given set of requirements. The steps 1-3 are the steps that are already used in current state-of-the-practise tools, the steps 4, 4a, 4b and 4c describe in further detail the functionality provided by our framework.

Step 1: The initial step consists of entering the requirements for the product in some natural language-like format, yet using some restricted grammar that avoids ambiguity and improves understanding.

Step 2: In the next step the requirements are formalized by automatically translating into predefined temporal logic, e.g., LTL. Usually, each particular grammar used in the previous step is tailored for subsequent translation into one particular temporal logic.

Step 3: Based on the type of used temporal logic a suitable tool for consistency checking is selected and the set of formalized requirements C is checked for consistency. If the set is consistent, then the software development process can proceed into a next stage. Otherwise, the source of the inconsistency should be identified and the requirements should be fixed. The steps 1-3 can be performed in a semi-automated way by employing some of the existing requirements authoring tools. In particular, the steps 2-3 are usually fully automated, however the first step requires an interaction with a user. The requirements authoring tools usually supply the user with an automatic grammar checking and even suggest her words that can be used to complete unfinished sentences.

Step 4: The inconsistent set of requirements C is submitted to our framework. The core of the framework is an online MIS enumeration algorithm which is applicable to any type of formalized requirements; the only condition is that a consistency checking tool for that particular format exists. Basically any domain agnostic algorithm for MISes enumeration can be used in here, however, as we focus on temporal logics, we plan to use TOME which was tailored for this type of constraints. Complete description of the TOME algorithm is available in [9]; briefly speaking TOME iteratively repeats the following steps:

Step 4a: Choose some subset S of C such that the consistency of S is not known yet and check it for consistency. If S is shown to be inconsistent then continue with the step 4b, otherwise repeat the step 4a.

Step 4b: Find a MIS contained in S. A universal way how to do it is to iteratively remove one-by-one requirement from S, after each removal check S for consistency and put back every requirement whose removal caused S to be consistent. Once we try to remove each requirement from S in this way, the remaining set is a MIS of S and thus also a MIS of C. This simple approach can be used for every formal specification language as it relies only on the existence of a consistency checking tool. However it is very inefficient as it performs too many consistency checks. Therefore, we would like to develop specialized single MIS extracting approaches for particular languages; mainly for LTL formulae.

Step 4c: Based on the found MIS, determine (in)consistency of other subsets of C. If there remains a subset whose

consistency is not known yet, then continue with the step 4a.

Because TOME enumerates MISes incrementally, we can either let it to enumerate all MISes, or stop it once a sufficient amount of MISes is outputted; this decision is carried out by the user of the framework.

The MISes outputted by our framework can be subsequently used to fix the inconsistencies among the requirements. The whole process is usually repeated several times till the set of requirements is found both to be consistent and providing sufficient coverage of user expectations.

4.2 Evaluation

We are currently participating in the AMASS project. The goal of AMASS is to create and consolidate the de-facto European-wide open tool platform, ecosystem, and self-sustainable community for assurance and certification of Cyber-Physical Systems (CPS) in the largest industrial vertical markets including automotive, railway, aerospace, space, energy. A need for identification of sources of inconsistency of an inconsistent set of requirements is one of the issues that is being dealt with in AMASS. The framework resulting from our work will be practically evaluated by our industrial partners in the AMASS project. This means that a real set of requirements taken from industry will be used as benchmarks. Also, a feedback from requirements engineers will be obtained.

5 RESEARCH STATUS

This work is a part of my PhD studies. Besides the problem of finding MISes in the domain of temporal logic we also focus on solving the problem of MISes enumeration in other domains. We have conducted a survey of both single and multiple MIS enumeration approaches in SAT and SMT domains and we have identified techniques that can be used, to some extent, also in the domain of temporal logics. As for the high-level architecture schema of the framework presented in this paper (Fig. 2) there are publicly available tools for performing the steps 1-3. As for the step 4, we have developed a domain agnostic algorithm, called TOME, for online MISes enumeration. In order to finish the framework we need to conduct a survey of consistency checking tools and also develop some domain specific single MIS extraction algorithms that would be used as TOME's subroutines.

6 CONCLUSION

In this work we proposed a framework for identifying minimal inconsistent subsets (MISes) of a given inconsistent set of requirements. The core part of this framework, i.e., a domain agnostic algorithm for online enumeration of MISes, has been already developed. Future work will focus on a development of a domain specific single MIS extraction algorithms that are used as subroutines of the domain agnostic algorithm. The resultant implementation of the framework will be evaluated by our industrial partners in the AMASS project.

ACKNOWLEDGEMENT

This project has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 692474, project name AMASS. This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Czech Republic, Germany, Sweden, Austria, Italy, United Kingdom, France.

REFERENCES

[1] James Bailey and Peter J Stuckey. 2005. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *Practical Aspects of Declarative Languages*. Springer, 174–186.

[2] Jiří Barnat, Petr Bauch, Nikola Beneš, Luboš Brim, Jan Beran, and Tomáš Kratochvíla. 2016. Analysing sanity of requirements for avionics systems. *Formal Aspects of Computing* doi: 10.1007/s00165-015-0348-9 (2016). DOI: http://dx.doi.org/10.1007/s00165-015-0348-9

[3] Jiri Barnat, Petr Bauch, and Lubos Brim. 2012. Checking Sanity of Software Requirements. In *SEFM (Lecture Notes in Computer Science)*, Vol. 7504. Springer, 48–62.

[4] Jiri Barnat, Jan Beran, Lubos Brim, Tomas Kratochvila, and Petr Rockai. 2012. Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs. In *FMICS (Lecture Notes in Computer Science)*, Vol. 7437. Springer, 78–92.

[5] Jiri Barnat, Lubos Brim, Vojtech Havel, Jan Havlícek, Jan Kriho, Milan Lenco, Petr Rockai, Vladimír Still, and Jirí Weiser. 2013. DiVinE 3.0 - An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *CAV (Lecture Notes in Computer Science)*, Vol. 8044. Springer, 863–868.

[6] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. 2001. Efficient Detection of Vacuity in Temporal Model Checking. *Formal Methods in System Design* 18, 2 (2001), 141–163.

[7] Anton Belov and João Marques-Silva. 2012. MUSer2: An Efficient MUS Extractor. *JSAT* 8, 3/4 (2012), 123–128.

[8] Jaroslav Bendík, Nikola Beneš, Jiří Barnat, and Ivana Černá. 2016. Finding Boundary Elements in Ordered Sets with Application to Safety and Requirements Analysis. In *SEFM (Lecture Notes in Computer Science)*, Vol. 9763. Springer, 121–136.

[9] Jaroslav Bendík, Nikola Benes, Ivana Cerná, and Jiri Barnat. 2016. Tunable Online MUS/MSS Enumeration. In *FSTTCS (LIPIcs)*, Vol. 65. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 50:1–50:13.

[10] Matteo Bertello, Nicola Gigante, Angelo Montanari, and Mark Reynolds. 2016. Leviathan: A New LTL Satisfiability Checking Tool Based on a One-Pass Tree-Shaped Tableau. In *IJCAI*. IJCAI/AAAI Press, 950–956.

[11] Stefan Blom, Wan Fokkink, Jan Friso Groote, Izak van Langevelde, Bert Lisser, and Jaco van de Pol. 2001. μCRL: A Toolset for Analysing Algebraic Specifications. In *CAV (Lecture Notes in Computer Science)*, Vol. 2102. Springer, 250–254.

[12] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *CAV (Lecture Notes in Computer Science)*, Vol. 2404. Springer, 359–364.

[13] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 2001. *Model checking*. MIT Press.

[14] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. 2016. Spot 2.0 - A Framework for LTL and ω -Automata Manipulation. In *ATVA (Lecture Notes in Computer Science)*, Vol. 9938. 122–129.

[15] François Hantry and Mohand-Said Hacid. 2011. Handling Conflicts in Depth-First Search for LTL Tableau to Debug Compliance Based Languages. In *FLACOS (EPTCS)*, Vol. 68. 39–53.

[16] Mike Hinchey, Michael Jackson, Patrick Cousot, Byron Cook, Jonathan P. Bowen, and Tiziana Margaria. 2008. Software engineering and formal methods. *Commun. ACM* 51, 9 (2008), 54–59.

[17] Jianwen Li, Shufang Zhu, Geguang Pu, and Moshe Y. Vardi. 2015. SAT-Based Explicit LTL Reasoning. In *Haifa Verification Conference (Lecture Notes in Computer Science)*, Vol. 9434. Springer, 209–224.

[18] Mark H. Liffiton, Alessandro Previti, Ammar Malik, and Joao Marques-Silva. 2015. Fast, flexible MUS enumeration. *Constraints* (2015), 1–28.

[19] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. 2014. Accelerated Deletion-based Extraction of Minimal Unsatisfiable Cores. *JSAT* 9 (2014), 27–51.

[20] Amir Pnueli. 1977. The Temporal Logic of Programs. In *FOCS*. IEEE Computer Society, 46–57.

[21] Kristin Y. Rozier and Moshe Y. Vardi. 2011. A Multi-encoding Approach for LTL Symbolic Satisfiability Checking. In *FM (Lecture Notes in Computer Science)*, Vol. 6664. Springer, 417–431.

[22] Viktor Schuppan. 2012. Towards a notion of unsatisfiable and unrealizable cores for LTL. *Sci. Comput. Program.* 77, 7-8 (2012), 908–939.

[23] Viktor Schuppan and Luthfi Darmawan. 2011. Evaluating LTL Satisfiability Solvers. In *ATVA (Lecture Notes in Computer Science)*, Vol. 6996. Springer, 397–413.