Automata Based Test Generation with SpecPro

Simone Vuotto[*][†], Massimo Narizzano[†], Luca Pulina[*], Armando Tacchella[†]
[*]Chemistry and Farmacy Dept., University of Sassari, Italy
[†]DIBRIS, University of Genoa, Italy
{svuotto,lpulina}@uniss.it, {massimo.narizzano,armando.tacchella}@unige.it

***Abstract*—In this paper we introduce a new automata based test generation algorithm implemented in SpecPro, our library for supporting analysis and development of formal requirements in cyber-physical systems. We consider specifications written in Linear Temporal Logic (LTL) from which we extract automatically trap properties representing the expected behaviour of the system under development. With respect to manual generation, the main advantage of SpecPro is that it frees the developer from the burden of generating tests in order to achieve stated coverage targets. Our goal is to have SpecPro handle specifications of small-but-critical components in an effective way.**

***Index Terms*—Requirements Engineering, Automatic Test Generation, Cyber-Physical Systems**

I. Context and Motivation

Writing test cases early in the design process is a good practice, advocated by methodologies such as Test-Driven Development (TDD). Although there is no general consensus on the overall TDD process and benefits [1], the influence of early testing in improving design and software quality is widely acknowledged. Moreover, the test-first principle can help in identifying and resolving requirements issues and under-specified behaviors early in the process.

Our goal is to automatically extract a test suite from the requirement specification, giving the user a tool to systematically analyze the behaviors described in the specification and to put them to work in the subsequent phases. In order to extract test cases from requirements, different requirement-based coverage metrics have been proposed in the literature, although they usually rely on a complete model of the system for test generation (see [2] for a survey of available methods). In particular, the work here presented takes inspiration from [3], a linear temporal logic (LTL) [4] specification-based test-case generation methodology. They present some coverage criteria over the Büchi Automata representation of LTL requirements, and then explore the automata in order to extract *trap properties*, i.e., automata comprised of a finite prefix and a suffix that repeats infinitely often. The negation of trap-properties can be checked against a complete model of the system; if a counter-example is returned then it is considered as a test, otherwise the model does not implement the behaviour. This methodology saves the test engineer from the burden of writing tests manually, so it could be used to boost TDD. However, it relies on the availability of a complete model which, in a TDD context, we might simply not have, and incomplete models would not suffice. Moreover, it also relies on the fact that the model satisfies all the requirements, i.e., each requirement has to be verified against the model.

The research of Luca Pulina and Simone Vuotto has been funded by the EU Commission's H2020 Programme under grant agreement N.732105 (CERBERO). The research of Luca Pulina has been also partially funded by the Sardinian Regional Project PROSSIMO (POR FESR 2014/20-ASSE I) and the FitOptiVis (ID: 783162) project.

In this paper, we extend the contribution of [3], presenting a new way to generate test-cases that avoid relying on the existence of a complete model of the system. In particular, while in [3] a different automaton is built for each behavior, we build a single automaton representing all behaviors of the specification and we traverse it in order to extract valid lasso-shaped test cases. This approach frees us from the need of a model, but we require a complete specification to be known in advance. In this regard, we are more closely related to the synthesis problem, but we limit our scope to the generation of a limited set of behaviors that the final system should implement, and we do not aim at synthesizing the whole system. Since the generation of the automaton can be costly, we consider our approach effective for small-but-critical subsystems for which the speed of development must be balanced with high confidence in their correctness. Finally, we contribute SpecPro, an open-source tool that implements the proposed algorithm, along with other features, such as the consistency checking of requirements and the identification of inconsistent minimal sets thereof.

The rest of the paper is organized as follows. In Section II we give some definitions that are used in Section III to describe the general idea underlying the test generation algorithm. In Section IV we present the tool SpecPro and briefly describe its functionalities, while in Section V we show an example to illustrate how the algorithm works. Finally, Section VI concludes the paper with some final remarks.

II. Background

LTL formulae consist of atomic propositions, Boolean operators, and temporal operators. The syntax of a LTL formula ϕ is given as follows:

$$\phi = \top \mid \bot \mid a \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \mathcal{X}\,\phi_1 \mid \phi_1\,\mathcal{U}\,\phi_2 \mid (\phi)$$

where $a \in AP$, ϕ, ϕ_1, ϕ_2 are LTL formulae, $\mathcal{X}$ is the "next" operator and $\mathcal{U}$ is the "until" operator. We also consider other Boolean connectives like "$\wedge$" and "$\rightarrow$" with the usual meaning and the temporal operators $\Diamond\,\phi$ ("eventually") to denote $\top\,\mathcal{U}\,\phi$ and $\Box\,\phi$ ("always") to denote $\neg\Diamond\,\neg\phi$. In the following, unless specified otherwise using parentheses, unary operators have higher precedence than binary operators. Briefly, the semantics of an LTL formula ϕ yields a ω-language *Words*(ϕ) of infinite words satisfying ϕ, *i.e.*, infinite sequences over the 2^{AP} alphabet (see [5] for a full description).

Definition 1 (Non Deterministic Büchi Automata): A non deterministic Büchi Automata (NBA) $\mathcal{A}$ is a tuple $\mathcal{A}$ = (Q, Σ, δ, q_0, F) where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \rightarrow 2^Q$ is a transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is a set of accept states, called acceptance set. Σ^ω denotes the set of all infinite words over the alphabet Σ. We denote $\sigma = A_0A_1A_2 \ldots \in \Sigma^\omega$ one such word and $\sigma[i] = A_i$ for the i-th element of σ. For sake of simplicity, the transition relation $q' \in \delta(q,A)$ where q, $q' \in$ Q, and $A \in \Sigma$, can be rewritten as : $q \xrightarrow{A} q'$.

In Figure 1 is presented an example of Büchi Automata, where Q = {0,1,2,3}, Σ = $2^{\{a,b,c\}}$, Q_0 = {0}, F={1}, and transition of the form $q_i \xrightarrow{a \vee b} q_{i+1}$ is a short notation for the three transitions $q_i \xrightarrow{a} q_{i+1}$, $q_i \xrightarrow{a,b} q_{i+1}$, $q_i \xrightarrow{b} q_{i+1}$

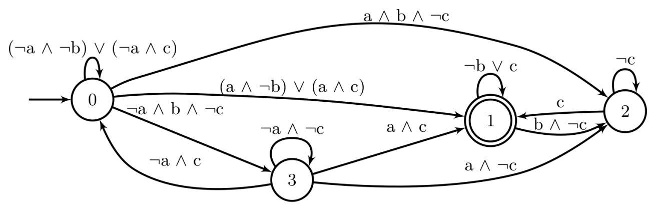

Fig. 1. Büchi Automata example.

Definition 2 (Run): A run for a NBA $\mathcal{A}$ = (Q, Σ, δ, q_0, F) is a an infinite sequence ϱ= $q_0q_1q_2$... of states in $\mathcal{A}$ such that q_0 is the initial state and $q_{i+1} \in \delta(q_i, A_i)$ for some $A_i \in \Sigma$. Given a run ϱ, we define *Words*(ϱ) the set of words that can be produced following the transitions in ϱ.

Definition 3 (Induced Run): Given a word σ, a run ϱ is said to be induced by σ, denoted $\sigma \vdash \varrho$, iff $q_{i+1} \in \delta(q_i, \sigma[i])$ for all $i \geq 0$.

Definition 4 (Accepting run): A run ϱ is accepting if there exist $q_i \in F$ such that q_i occurs infinitely many times in ϱ. We denote $acc(\mathcal{A})$ the set of accepting runs for $\mathcal{A}$.

Definition 5 (Lasso-Shaped run): A run ϱ over a NBA $\mathcal{A}$ = (Q, Σ, δ, q_0, F) is lasso-shaped if it has the form ϱ=$\alpha(\beta)^\omega$, where α and β are finite sequences over the states Q. A lasso-shaped run is also accepting if $\beta \cap F \neq \emptyset$.

The length of ϱ is defined as $|\varrho|$= $|\alpha|$ + $|\beta|$, where $|\alpha|$ (resp. $|\beta|$) is the length of the finite sequence of states α (resp. β).

Definition 6 (ω-language recognized by $\mathcal{A}$): A ω-language $\mathcal{L}(\mathcal{A})$ of a NBA $\mathcal{A}$ = (Q, Σ, δ, q_0, F) is the set of all infinite words that are accepted by $\mathcal{A}$. A word $\sigma \in \Sigma^\omega$ is accepted by $\mathcal{A}$ iff there exists an accepting lasso-shaped run ϱ of $\mathcal{A}$ induced by σ. Formally, $\mathcal{L}(\mathcal{A}) = \{\sigma \in \Sigma^\omega \mid \exists \varrho = \alpha(\beta)^\omega . \varrho \in acc(\mathcal{A}) \wedge \sigma \vdash \varrho\}$.

III. Automatic Test Generation from Büchi Automata

Given the set of requirements written as LTL formulae φ_1, ..., φ_n, over the set of symbols AP, we build the conjunctive formula $\Phi = \varphi_1 \wedge ... \wedge \varphi_n$, and construct a NBA $\mathcal{A}_\Phi$ such that $\Sigma = 2^{AP}$ and the language $\mathcal{L}(\mathcal{A}_\Phi)$ contains all and only the words in *Words*(Φ). The automaton $\mathcal{A}_\Phi$ represents the allowed behaviors described by the requirements. The algorithm aims to traverse $\mathcal{A}_\Phi$, in order to look for lasso-shaped accepting runs with length K that produce a set of words $W \subseteq \mathcal{L}(\mathcal{A}_\Phi)$. Notice that the size of the set of accepting runs in $\mathcal{A}_\Phi$ with length K increases with K and K is fixed a priori. Furthermore, inspired by testing techniques, we introduce a testing objective O as a set of elements that must be satisfied by a set of lasso-shaped runs, so that SpecPro generates a corresponding set of words $W \subset$ *Words*(Φ) that satisfy as many elements as possible in O. In particular we define different testing objectives, i.e. coverage measures:

- STATE COVERAGE: each state in the NBA should be traversed at least once in a run induced by a generated word.
- ACCEPTANCE COVERAGE: each acceptance state in the NBA should be traversed at least once in the recurring part of the lasso-shaped run induced by a generated word.
- TRANSITION COVERAGE: each transition in the NBA should be traversed at least once in a run induced by a generated word.

These coverage measures are similar to the ones proposed in [3]. Since we only have one automaton, we do not need to distinguish between weak and strong coverage. Moreover, our acceptance coverage is the NBA version of the *Accepting State Combination Coverage* (defined over Generalized Büchi Automaton).

The transition coverage clearly subsumes the state coverage, i.e., in order to cover all the transitions of an NBA, also all the states must be covered. On the other hand, the acceptance condition is orthogonal to the other two conditions, i.e., state and transition coverage do not guarantee acceptance coverage, and viceversa. Consequently, we also add the combined coverage of acceptance plus state and transition coverage respectively. Testing objectives are also used to stop the test generation; when all the testing objectives have been covered, the generation should stop.

```
function NBA2WORDS($\mathcal{A}$,$O$,$K$)
    $\mathcal{W} \leftarrow \emptyset$ ,
    while $O \neq \emptyset$ do
        $o \leftarrow$ RandomPick($O$)                      ▷ Objective that must be satisfied
        $\varrho \leftarrow$ GetAcceptedRunLS($\mathcal{A}$,$o$,$K$)
        if $\varrho = \emptyset$ then                       ▷ The objective can not be satisfied
            $O \leftarrow O \setminus \{o\}$
        else
            $\sigma \leftarrow$ GetWord($\mathcal{A}$,$\varrho$)
            $\mathcal{W} \leftarrow \mathcal{W} \cup \{\sigma\}$
            for all $o'$ satisfied by $\sigma$ do
                $O \leftarrow O \setminus \{o'\}$
            end for
        end if
    end while
    return $\mathcal{W}$                          ▷ Returns lasso shaped words accepted by $\mathcal{A}$
end function
```

Fig. 2. Infinite Word Generator Algorithm for NBA.

In Figure 2 we present the algorithm implemented in SpecPro for infinite word generation of bound K. The procedure NBA2WORDS generates a set $\mathcal{W} \subset$ *Words*(Φ) with $\Phi = \varphi_1 \wedge ... \wedge \varphi_n$ obtained from the set of requirements $R = \{\varphi_1, ..., \varphi_n\}$. The input is a NBA $\mathcal{A}_\Phi$ obtained from the formula Φ, a testing objective O and the size K of the lasso-shaped runs. We implemented the generation of the NBA using *Spot* [6], a C++ library for LTL, ω-automata manipulation and model checking. NBA2WORDS starts by picking an element from O (if not empty) (line 4), and then it calls a function GETACCEPTEDRUNLS that traverses $\mathcal{A}$ to look for an accepting lasso-shaped run ϱ with length K (line 5). An empty ϱ means that there is no lasso-shaped accepting run with length K satisfying the objective o (line 6). On the other hand, if such a ϱ exists, the algorithm calls GETWORD to compute the word σ induced from ϱ; σ is added to the set of words $\mathcal{W}$ (lines 9–10), and then each element o' satisfied by σ is eliminated from O (line 11). This is repeated until there are no more elements in O to satisfy. Notice that NBA2WORDS does not ensure that $\mathcal{W}$ satisfies all the elements of O; indeed, there could be an element that cannot be satisfied by any run with length K, but it can be satisfied increasing the length of the words generated.

The interpretation of the generated test cases depends on the level of abstraction of requirements and on the system under test (SUT). They can be directly interpreted as tests for the SUT, or further refined with classical model-checking-based strategies if a (partial) model of the system is available. For example, in our experiments we converted lasso-shaped words in never-claims and extracted counterexamples with SPIN [7], extending the initial word with a refined behavior. In order to solve the lasso-shaped part of the test, we can apply the same strategies described in [8] either for white- or black-box testing.

IV. SPECPRO

SPECPRO is an open-source[1] Java library for supporting analysis and development of formal requirements. It takes in input a list of requirements expressed in textual form as Property Specification Patterns (PSPs) [9] or as LTL formulae; and produces different outputs, depending on the specific task.

PSPs are meant to describe the essential structure of system behaviors in form of structured English sentences [10] and to provide expressions of such behaviors in a range of common formalisms. An example of a PSP is given in Figure 3 — with some part omitted for sake of readability.[2] In more detail, a PSP is composed of two parts: (i) the *scope*, and (ii) the *body*. The *scope* is the extent of the program execution over which the pattern must hold, and there are five scopes allowed: *Globally*, *Before*, *After*, *Between*, *After-until*. The *body* of a pattern describes the behavior that we want to specify.

In particular, SPECPRO contains a collection of algorithms and data structures to accomplish the following tasks:

[1]https://gitlab.sagelab.it/sage/SpecPro

[2]The full list of PSPs considered in this paper and their mapping to LTL and other logics is available at http://ps-patterns.wikidot.com/.

a) Encode into a LTL satisfiability problem: it maps each PSP into the equivalent LTL formula, encodes the specification into an LTL satisfiability problem and translates it for a specific model-checker input. Currently AALTA [11], NUSMV [12] PLTL [13], and TRP++ [14] are supported. SPECPRO implements the encoding presented in [15], whereby PSPs are extended by considering Boolean as well as atomic numerical assertions of the form $x \bowtie c$, where x is a variable of the system, $c \in \mathbb{R}$ is a constant real number and the operator $\bowtie \in \{<, <=, =, >=, >\}$ has the usual interpretation.

b) Consistency Checking: it performs an inner consistency check of the requirements set, i.e., logical errors in the specification that prevent any possible system to satisfy all requirements. It employs the LTL satisfiability encoding described before, it handles an external call to the model checker, and interprets the returned output, providing a user-friendly API for the developer (currently only AALTA and NUSMV are supported).

c) Minimal Unsatisfiable Core (MUC) extraction [16]: In case of an inconsistent specification, the user often needs an hint about the cause of the inconsistency. Usually, in order to help the user, a Minimal Unsatisfiable Core (MUC) [17], namely an irreducible subset of requirements that is still inconsistent, is extracted. SPECPRO implements two algorithms for MUCs extraction from a PSP specification document: *(i)* the common linear deletion-based algorithm, that removes requirements one at a time and test the satisfiability of the reduced set; and *(ii)* the dichotomic one, that employs a greedy strategy and it is usually faster when the MUC is much smaller then the full requirements set. These algorithms build on top of the components developed for consistency checking.

d) Automatic Test Generation: It implements the algorithm described in Section III. It relies on SPOT to build a Büchi Automaton of the specification and employs the Iterative Deepening Deep First Search (IDDFS) algorithm to extract valid paths of different lengths that can be used for testing. The user can define a minimum and a maximum value of K and a coverage criteria. The algorithm then starts searching for small values of K, iteratively increasing it until the testing objective is fulfilled or the maximum value of K is reached.

Finally, SPECPRO also provides a minimal command-line interface that enables the user to perform the same tasks from a shell. A user-friendly graphical user interface is also available with REQV [18], a web application that builds on top of SPECPRO to help non-expert users in the consistency checking of requirements.

V. EXAMPLE

In this section we present an example to illustrate how the algorithm works. Consider the following requirements written in LTL:

R1 $\Box\ (b \rightarrow \Diamond\ c)$ **R2** $\Diamond\, a$

Running SPECPRO on this specification, with $K_{min} = 2$, $K_{max} = 5$, and State Coverage as criteria, the following steps are executed:

Response

Describe cause-effect relationships between a pair of events/states. An occurrence of the first, the cause, must be followed by an occurrence of the second, the effect. Also known as Follows and Leads-to.

Structured English Grammar

It is always the case that if P holds, then S eventually holds.

Example

It is always the case that if object_detected holds, *then* moving_to_target *eventually holds.*

Fig. 3. Response Pattern. A pattern is comprised of a *Name*, an (informal) statement describing the behavior captured by the pattern, and a (structured English) statement that should be used to express requirements.

- Build the conjunctive formula $\phi = \Box\ (b \rightarrow \Diamond\ c) \wedge \Diamond a$.
- Set the starting bound for IDDFS to $K = 2$.
- Build the NBA of ϕ (shown in Figure 1) with SPOT.
- Generate the accepting lasso-shaped run $\rho_1 = \{0\}\{1\}^\omega$, inducing the word $\omega_1 = a, \{c\}^\omega$.
- Update O with ω_1, marking states 0 and 1 as covered. Now only states 2 and 3 remain to be covered.
- Since no other lasso shaped accepting run of length 2 can be generated, the bound K is increased to 3.
- Several lasso shaped runs with length 3 can be generated, but here we are only interested in those satisfying testing objectives that no previous generated words can satisfy. For example, the run $\rho_2 = \{0\}\{0\}\{1\}^\omega$, inducing the word $\omega_2 = \overline{ab}, ac, \{c\}^\omega$. However no new testing objective is satisfied by ω_2, so it is discarded.
- Proceeding further, two other lasso shaped runs are found: (i) $\rho_3 = \{0\}(\{1\}\{2\})^\omega$, inducing the word $\omega_3 = ac,\{b\overline{c},c\}^\omega$, and (ii) $\rho_4 = \{0\}\{3\}(\{1\})^\omega$, inducing the word $\omega_3 = \overline{a}b\overline{c}, ac, \{\mathrm{c}\}^\omega$. The two words satisfy all the testing objectives, namely covering states 2 and 3 since the runs traverse them.
- No more states have to be visited, so the algorithm terminates successfully. Three different words are generated, indicating the behaviors wherefrom tests can be extracted.

VI. Conclusion and Future Work

In this paper we presented a new automata based test generation procedure and related implementation in the SPECPRO library. To carry on the test generation task, requirements are formalized as LTL formulas and the Büchi Automaton representation of their conjunction is built. The automaton is then explored with different strategies to extract words that have to be implemented in the system. Concerning current and future work, our next steps will focus on investigating additional coverage criteria and quality measurements to assess the generated words. Moreover, we wish to deal with the scalability issues we encountered during our preliminary experiments. Finally, SPECPRO is still under active development and we aim at adding new functionalities and explore more expressive logics.

References

[1] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo, "A dissection of the test-driven development process: does it really matter to test-first or to test-last?" *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 597–614, 2017.

[2] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: a survey," *Software Testing, Verification and Reliability*, vol. 19, no. 3, pp. 215–261, 2009.

[3] B. Zeng and L. Tan, "Test reactive systems with büchi-automaton-based temporal requirements," in *Theoretical Information Reuse and Integration*. Springer, 2016, pp. 31–57.

[4] A. Pnueli and Z. Manna, "The temporal logic of reactive and concurrent systems," *Springer*, vol. 16, p. 12, 1992.

[5] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[6] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu, "Spot 2.0 — a framework for LTL and ω-automata manipulation," in *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, ser. Lecture Notes in Computer Science, vol. 9938. Springer, Oct. 2016, pp. 122–129.

[7] G. J. Holzmann, *The SPIN model checker: Primer and reference manual*. Addison-Wesley Reading, 2004, vol. 1003.

[8] L. Tan, O. Sokolsky, and I. Lee, "Specification-based testing with linear temporal logic," in *Conference on Information Reuse and Integration*, 2004, pp. 483–498.

[9] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st International conference on Software engineering*, 1999, pp. 411–420.

[10] S. Konrad and B. H. Cheng, "Real-time specification patterns," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 372–381.

[11] J. Li, Y. Yao, G. Pu, L. Zhang, and J. He, "Aalta: an LTL satisfiability checker over infinite/finite traces," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 731–734.

[12] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," in *14th International Conference on Computer Aided Verification (CAV 2002)*, 2002, pp. 359–364.

[13] S. Schwendimann, "A new one-pass tableau calculus for PLTL," in *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer, 1998, pp. 277–291.

[14] U. Hustadt and B. Konev, "TRP++ 2.0: A temporal resolution prover," in *19th International Conference on Automated Deduction*, 2003, pp. 274–278.

[15] M. Narizzano, L. Pulina, A. Tacchella, and S. Vuotto, "Consistency of property specification patterns with boolean and constrained numerical signals," in *NASA Formal Methods: 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings*, vol. 10811. Springer, 2018, pp. 383–398.

[16] ——, "Property specification patterns at work: Verification and inconsistency explanation," *Innovations in Systems and Software Engineering*, To appear.

[17] M. H. Liffiton and K. A. Sakallah, "Algorithms for computing minimal unsatisfiable subsets of constraints," *Journal of Automated Reasoning*, vol. 40, no. 1, pp. 1–33, 2008.

[18] S. Vuotto, M. Narizzano, L. Pulina, and A. Tacchella, "Poster: Automatic consistency checking of requirements with reqv," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, To Appear.