Formal Specifications using Linear Temporal Logic

Simone Vuotto
[1]Universit'a degli Studi di Genova
{simone.vuotto@edu.unige.it}
[2]Universit'a degli Studi di Sassari
{svuotto@uniss.it}

ABSTRACT: *The definition of a formal specification is a fundamental step in the design of safety-critical systems. The specification defines the behaviors and constraints that the system under development is required to satisfy. Depending on the system dimension, the specification can be used to verify its correctness or the correctness of its abstract model. In some cases, the system can also be directly synthesized from the specification. However, when the system is too complex for a full verification or for synthesis, developers usually rely on testing to demonstrate the correct operation of the system. In this paper we present our ongoing work on automatic test cases generation, relying on Linear Temporal Logic (LTL) as a specification formalism. The presented algorithm is implemented in SPECPRO, our library for supporting analysis and development of formal requirements in cyber-physical systems.*

Keywords: Formal Specifications, Lineare Temporal Logic

Received: 10 September 2019, Revised 2 December 2019, Accepted 13 December 2019

DOI: 10.6025/jio/2020/10/1/13-21

1. Introduction

In the context of reactive systems, i.e., systems that maintain an ongoing interaction with the environment and react to external stimuli, it is important to check the correctness of their behavior over time. Formal verification techniques provides a viable solution to automatically check the system against a given specification, greatly increasing the confidence of their correctness. However, these techniques suffer of known scalability issues and the complete verification of the specification becomes impractical or even impossible for complex systems. For this reason, testing is the preferred technique for hardware and software verification in industry, although it provides less guarantees; testing can only detect the presence of errors, not their absence. Nonetheless, a formal specification can still be of great practical use to automatically generate test suites to show conformance of the model and the actual implementation, or, just to derive "interesting" test cases to check the developed system [2].

In this paper, we present a new algorithm 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 [6] for a survey of available methods). In particular, the proposed approach takes inspiration from [13], a linear temporal logic (LTL) [11] specification-based test-case generation methodology, and extend our previous work in [12].

Compared to [13], where a different automaton is built and checked against the model for each behavior, we build a single automaton representing all behaviors of the specification and we traverse it in order to extract valid 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. With respect to [12], we propose a different exploration strategy of the automaton and we introduce the concept of input and output sequences, that are used to generate the test cases. Finally, the proposed algorithm is implemented in the SPECPRO open-source library, along with other features, such as the consistency checking of requirements and the identification of inconsistent minimal sets thereof [9, 10].

The rest of the paper is organized as follows. In Section 2 we present some basic notions and definitions that are used in Section 3 to describe the proposed algorithm. Section 4 presents the experiments we made to illustrate how the algorithm works and to evaluate the generated test suite. Finally, Section 5 concludes the paper with some final remarks.

2. Background

LTL formulae consist of atomic propositions AP, 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, $\phi_2$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 $\omega-$ language Words (ϕ) of infinite words satisfying ϕ, i.e., infinite sequences over the 2^{AP} alphabet (see [1] for a full description).

Definition 1: (Non Deterministic Büchi Automaton). A non deterministic Büchi Automaton (NBA) $\mathcal{A}$ is a tuple $A = (Q, \Sigma, \delta, q_0, F)$ 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 Automaton, where $Q = \{0, 1, 2, 3\}$, $\Sigma = 2^{\{a, b, c\}}$, $Q_0 = \{0\}$, $F=\{1\}$, and transition of the form $q_i \xrightarrow{a\vee b} q_{i+1}$ is a short notation for the three transitions $q_i \xrightarrow{a} q_{i+1}$, $q_i \xrightarrow{a,b} q_{i+1}$, $q_i \xrightarrow{b} q_{i+1}$

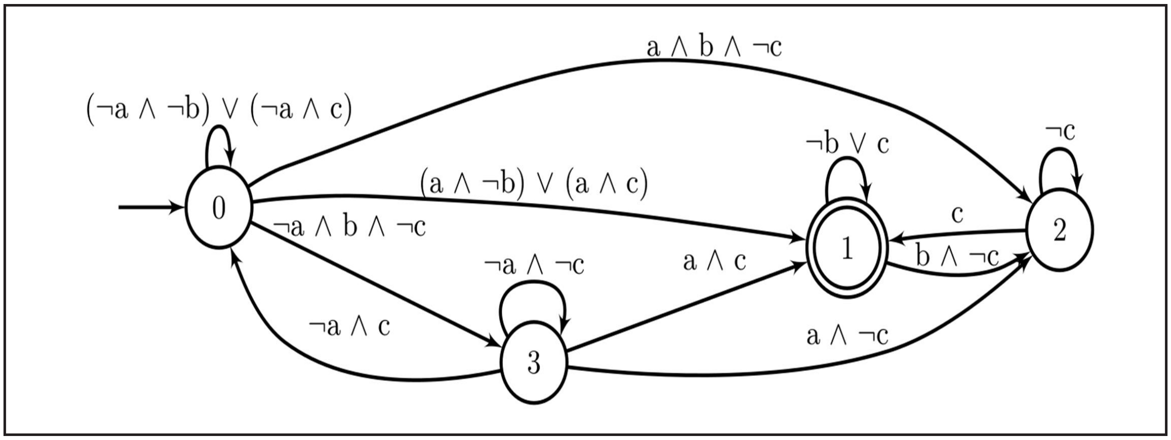

Figure 1. Buchi Automata example

Definition 2 (Run). A run for a $NBA\ \mathcal{A} = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$ is *a* an infinite sequence : $\varrho = q_0q_1q_2$...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**, ϱ run : 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). $\mathcal{A}$ run ϱ over a $NBA\ A = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$ is lasso-shaped if it has the form $\varrho = \alpha(\beta)^\omega$, 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 $\mathcal{L}$(A) of a $NBA\ A = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$ 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, $\mathcal{L}(\mathcal{A}) = \{\sigma \in \Sigma^\omega \mid \exists \varrho = \alpha(\beta)^\omega . \varrho \in acc(\mathcal{A}) \wedge \sigma \vdash \varrho\}$.

Transition System A transition system $\mathcal{T}$ is a tuple (T, t_0, τ) where T is a finite set of states, t_0 is the initial state and $\tau : T \times 2^I \rightarrow 2^O \times T$ is the transition function. The sets I and O are a partition of the atomic propositions AP that are controllable by the environment and by the system respectively. The transition function τ maps a state t and a valuation of the inputs $i \in 2^I$ to a valuation of the outputs, also called labeling, and a next state t'. If the labeling produced by $\mathcal{T}$ (t, i) is independent of i, we call $\mathcal{T}$ a state-labeled (or Moore) transition system and transition-labeled (or Mealy) otherwise. Most of the state of the art tools for reactive *LTL* synthesis, such as ACACIA+ [5], STRIX [8] or BOSY [4] use Mealy Machines to describe the reactive systems synthesized.

Given an infinite word $i_0i_1 \cdots \in (2^I)^\omega$ over the inputs, $\mathcal{T}$ can be traversed applying $\tau(t_j, i_j) = (o_j, t_{j+1})$ for every $j \geq 0$. The application of $\mathcal{T}$ for every input i_j, starting from t_0, produces an infinite trace $(t_0 \cup i_0 \cup o_0)(t_1 \cup i_1 \cup o_1) \cdots \in (2^{T \cup I \cup O})^\omega$. The projection of a trace to the atomic propositions is a path $w \in (2^{I \cup O})^\omega$. We denote the set of all paths generated by a transition system $\mathcal{T}$ as Paths(T). A transition system realizes an *LTL* formula φ if $Paths(\mathcal{T}) \subseteq \mathcal{L}(\varphi)$.

3. Automatic Test Case Generation

In this section, we describe a new methodology to automatically generate a set of test cases from a given LTL specification. We first give a general overview of the algorithm along with an example, and then in subsections 3.1 and 3.2 we describe the way we select input and output sequences respectively.

The algorithm, shown in Algorithm 1, takes in input a Buchi Automaton A_Φ of Φ and the sets I and O of input and output atomic propositions, respectively, such that $I \cup O = AP$ and $I \cap O = \emptyset$. In the general case of multiple LTL requirements $\phi_1, \ldots, \phi_n$, Φ is build as a conjunctive formula $\Phi = \phi_1 \wedge \cdots \wedge \phi_n$, while for a specification in TLSF [7] format, the formula is built with the SyFCo tool. The Buchi Automaton of the formula is built with Spot [3] v2.7.5 tool. The algorithm is divided into two steps: (i) we extract a set *Is* of suitable input sequences; (ii) for each input sequence σ_i we compute one or more output sequences that have to be checked for input σ_i.

Algorithm 1 Test Case Generation

```
function GENERATE ($\mathcal{A}_\Phi$, $I$, $O$)
Ts ← $\emptyset$;
```

```
    $Is$ ← FIND INPUT SEQUENCES ($\mathcal{A}_{\Phi}$, $I$)
    for $\sigma_i \in Is$ do
        $Os$ ← SELECT OUTPUTS ($\mathcal{A}_{\Phi}, \sigma_i, O$)
        $Ts$ ← $Ts$ [ $\{(\sigma_i, Os)\}$
    end for
    return $Ts$
end function
```

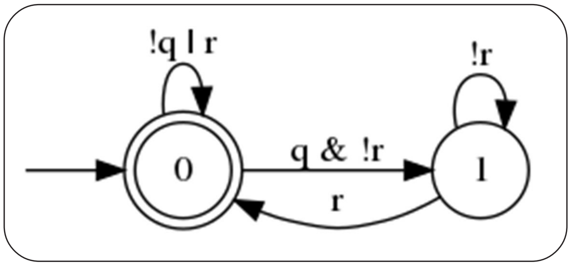

Figure 2. Buchi Automaton corresponding to the formula "$\Box\,(q \to \Diamond\, r)$"

The distinction between input and output sequences is needed because of the intrinsic non-determinism normally allowed by a LTL specification. In fact, a specification usually describes a (possibly infinite) range of allowed sequences. The situation is even worst if the system is under-specified, i.e., the behavior of the atomic propositions is loosely restricted.

For example, consider the formula $\phi = \Box\,(q \to \Diamond\, r)$, meaning that every query q must be followed by a response r. In this case q is an input variable and r is an output variable. The corresponding NBA is depicted in figure 2. Now, let's consider a three steps input sequence $\sigma = [\{\overline{q}\}, \{q\}, \{\overline{q}\}]$, where q is true only in the second step. If we check all possible prefixes of length 3 that are in Words (ϕ), given the assignment of q fixed by σ, we see that multiple solutions are possible. $[\{\overline{qr}\}, \{qr\}, \{\overline{qr}\}]$ is such a solution, and so are $[\{\overline{qr}\}, \{q\overline{r}\}, \{\overline{q}r\}]$ and $[\{\overline{qr}\}, \{qr\}, \{\overline{q}r\}]$. Also the sequence $[\{\overline{q}r\}, \{qr\}, \{\overline{q}r\}]$ is accepted by the language, where r is always true no matter what's the behavior of q. Finally, also the sequence in which r is always false is a valid prefix, because if extended can lead to words in which ϕ is eventually satisfied. However, the transition system realizing such requirement, will eventually implement only one of such behaviors. Therefore, we need to keep track of the relation between these words. In this context, we consider a test case to be successful if applying the input sequence to the system under test, the generated output is contained in the set of allowed output sequences.

3.1. Find Input Sequences

Algorithm 2 shows the steps implemented to find the set of input sequences. After the initialization of the *Is* and *visited sets*, the function generate Filtered BA (line 4) returns $\mathcal{A}^I_{\Phi}$, a filtered version of the given automaton. The new automaton $\mathcal{A}^I_{\Phi}$ has the same set of states Q, initial state q_0 and acceptance set F of $\mathcal{A}_{\Phi}$, but the alphabet $\Sigma^I = 2^I$ and the transition function $\delta^I = \delta(q, A \cap I)$ $\forall A \in \Sigma, q \in Q$ are built from the input atomic propositions I only. The resulting language $\mathcal{L}(\mathcal{A}^I_{\Phi})$ is therefore more abstract then $\mathcal{L}(\mathcal{A}_{\Phi})$ and contains only words with input variables. The new automaton is therefore explored with the following strategy:

– For each state q in $\mathcal{A}^I_{\Phi}$, we check every outgoing transition not yet explored (i.e., not traversed by an already generated sequence) that lead to a new state q';

– For every such transition (q, q') a run ρ is built in the following way: the shortest path (computed with the function sp at line 9 of Algorithm 2) from q_0 to q is concatenated to the shortest path to go from q' to the nearest acceptance state q^* (computed at line 8, with a classical breadth-first search strategy, within function findNearestAcceptanceState);

Algorithm 2 Find Input Sequences from $\mathcal{A}_\Phi$

```
1: function FIND INPUT SEQUENCES ($\mathcal{A}_\Phi, I$)
2:     $Is \leftarrow \emptyset$
3:     $visited \leftarrow \emptyset$
4:     $\mathcal{A}^I_\Phi \leftarrow$ GENERATE FILTERED BA($\mathcal{A}_\Phi, I$)
5:     for $q \in \mathcal{A}^I_\Phi.Q$ do
6:         for $q' \in \mathcal{A}^I_\Phi.\delta(q, .)$ do
7:             if $(q, q') \notin visited$ then
8:                 $q^* \leftarrow$ FIND NEAREST ACCEPTANCE STATE ($\mathcal{A}^I_\Phi, q'$)
9:                 $\rho \leftarrow$ SP($\mathcal{A}^I_\Phi, q0, q$) . SP($\mathcal{A}^I_\Phi, q', q^*$)
10:                $\sigma \leftarrow$ GET WORD ($\mathcal{A}^I_\Phi, \rho$)
11:                $Is \leftarrow Is \cup \sigma$
12:                for $i \leftarrow 0$ to $|\rho| - 1$ do
13:                    $visited \leftarrow visited \cup (\rho[i], \rho[i+1])$
14:                end for
15:            end if
16:        end for
17:    end for
18:    $Is' \leftarrow$ REDUCE($Is$)
19:    return $Is'$
20: end function
```

– We extract a word σ from ρ (line 10), we add it to the set of input sequences (line 11) and we mark each transition in ρ as visited (lines 12-14).

At the end of the process, we have a set is of input sequences, computed from the exploration of each state and transition of the automaton. The procedure is built with three goals in mind: (i) exercise every behavior contained in the automaton at least once; (ii) maintain the number of sequences small; and (iii) keep the sequences as short as possible. Finally, before returning the set of sequences, we call the reduce function that remove the sequences that are pre-fixes of other ones. For instance, if both $\sigma_1 = [\{\overline{r}\}, \{\overline{r}\}]$ and $\sigma_2 = [\{\overline{r}\}, \{\overline{r}\}, \{r\}]$ are generated, σ_1 can be removed because it is a prefix of σ_2.

3.2 Select Outputs

As mentioned before, for each input sequence σ_i, there could be one or more associated output sequences that the system can implement in order to fulfill the specification. In this section we analyze more in details how to select these sequences. First, remember that the sequences we are seeking to extract are words accepted by the automaton $\mathcal{A}_\Phi$ over the alphabet $\Sigma = 2^{AP}$, namely the power set of all the atomic propositions in Φ. A simple strategy could be to use the input sequence σ_i as a word of the automaton to find all induced runs, according to Definition 7 given in Section 2, and then use these runs to extract all accepted words. However, the problem with this definition is that it is intended for words that share the same alphabet of the automaton, while in this case σ_i is defined over the alphabet $\Sigma^I \subseteq \Sigma$. Therefore, we modify the notion of induced run over input sequences as follow:

Definition 7 (Input Induced Run). Given a *NBA* $A = (Q, \Sigma, \delta, q_0, F)$, with $\Sigma = 2^{I \cup O}$, and an input word σ defined over the alphabet $\Sigma^I = 2^I$, a run ϱ is said to be input induced by σ, denoted $\sigma \vdash^I \varrho$, iff $q_{i+1} \in \delta(q_i, A) : A \cap I = \sigma[i]$ for all $i \geq 0$.

With this new definition, we can now implement Algorithm 3. For each induced run, computed relying on the definition above, we check if they are lasso shaped and accepting (line 5). If they are, we extract the corresponding words and we filter them out (lines 6-9) so that they only contain output variables. The generated words are inserted in *Os* (line 10) and finally returned (line 14) when all runs have been evaluated.

As a final remark, Algorithm 3 computes for each input sequence σ_i all the corresponding lasso-shaped accepting output sequences of length $|\sigma_i|$. On the one hand, it could be possible to produce more output sequences, extracting longer runs or weakening the lasso-shaped accepting condition. On the other hand, one could also think to further reduce this set, filtering it out with some heuristics. For example, we could take into account only the runs that visit acceptance states more often, or the ones that reach an acceptance state first.

Algorithm 3 Select Output Sequences for input σ_i from $\mathcal{A}_\Phi$

```
function SELECT OTPUTS ($\mathcal{A}_\Phi, \sigma_i, O$)
    $Os \leftarrow \emptyset$
    $InducedRuns \leftarrow$ FIND INPUT INDUCEDRUNS ($\mathcal{A}_\Phi, \sigma_i$)
    for $\rho \in InducedRuns$ do
        if IS LASSOSHAPED AND ACCEPTING ($\mathcal{A}_\Phi, \rho$) then
            for $\sigma \in$ GETWORDS($\mathcal{A}_\Phi, \rho$) do
                for $i \leftarrow 0$ to $|\sigma|$ do
                    $\sigma[i] \leftarrow \sigma[i] \cap O$
                end for
                $Os \leftarrow Os \cup \sigma$
            end for
        end if
    end for
    return $Os$
end function
```

4. Experiments

Let us start our experimental evaluation by introducing the following specification as our running example, with $I = \{request\_0, request\_1\}$ and $O = \{grant\_0, grant\_1\}$ being the set of input and output variables, respectively:

$$\Box\,((grant\_0 \wedge \Box \neg\, request\_0) \rightarrow (\Diamond \neg\, grant\_0)) \tag{1}$$

$$\begin{aligned}&\Box\,((grant\_0 \wedge \chi\,(\neg\, request\_0 \wedge \neg\, grant\_0)) \rightarrow \\ &\qquad \chi\,(\neg\, grant\_0\ W\,(request\_0 \wedge \neg\, grant\_0))\end{aligned} \tag{2}$$

$$\Box\,((grant\_1 \wedge \Box \neg\, request\_1) \rightarrow (\Diamond \neg\, grant\_1)) \tag{3}$$

$$\begin{aligned}&\Box\,((grant\_1 \wedge \chi\,(\neg\, request\_1 \wedge \neg\, grant\_1)) \rightarrow \\ &\qquad \chi\,(\neg\, grant\_1\ W\,(request\_1 \wedge \neg\, grant\_1))\end{aligned} \tag{4}$$

$$\Box\,(\neg\, grant\_0 \vee \neg\, grant\_1) \tag{5}$$

$$\neg\, grant\_0\ W\,(request\_0 \wedge \neg\, grant\_0) \tag{6}$$

$$\neg\, grant\_1\ W\,(request\_1 \wedge \neg\, grant\_1) \tag{7}$$

$$\Box\,(request\_0 \rightarrow \Diamond\, grant\_0) \tag{8}$$

$$\Box\,(request\_1 \rightarrow \Diamond\, grant\_1) \tag{9}$$

The specification describes the full arbiter of two clients; it eventually issues a grant for each client if they send a request (see formulae (8) and (9)). The specification also states that the grant should not be issued to the two clients at the same time (see formula (5)), that if no further requests arrive it should stop issuing the grant (formulae (1) and (3)) and that no grant should be issued until new requests arrives (formulae (2), (4), (6) and (7)).

Given the above specification, the algorithm described in Section 3 generates the 30 tests cases presented in Table 1. The first column shows the generated input sequences (for the sake of space, *request*_0 and *request*_1 are replaced with r_0 and r_1 respectively) and the second column reports, for each input sequence, the corresponding number of generated output

Input Seq.	**# of Output Seq.**
$\{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}$	10
$\{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{r_0, \overline{r_1}\}$	3
$\{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	4
$\{\overline{r_0}, r_1\}, \{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	1
$\{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}$	4
$\{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{r_0, \overline{r_1}\}, \{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	21
$\{r_0, \overline{r_1}\}, \{\overline{r_0}, r_1\}, \{r_0, r_1\}, \{\overline{r_0}, \overline{r_1}\}$	1
$\{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{r_0, \overline{r_1}\}, \{r_0, \overline{r_1}\}$	12
$\{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{r_0, \overline{r_1}\}, \{r_0, \overline{r_1}\}$	6
$\{r_0, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	8
$\{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	10
$\{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, r_1\}, \{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	22
$\{r_0, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	2
$\{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{r_0, \overline{r_1}\}, \{r_0, r_1\}, \{\overline{r_0}, \overline{r_1}\}$	4
$\{r_0, \overline{r_1}\}, \{\overline{r_0}, r_1\}, \{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	7
$\{r_0, \overline{r_1}\}, \{\overline{r_0}, r_1\}, \{r_0, \overline{r_1}\}, \{r_0, \overline{r_1}\}$	1
$\{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}$	11
$\{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{r_0, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	4
$\{\overline{r_0}, r_1\}, \{r_0, \overline{r_1}\}, \{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}$	1
$\{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	6
$\{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	6
$\{r_0, \overline{r_1}\}, \{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	1
$\{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	10
$\{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	3
$\{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{r_0, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	4
$\{\overline{r_0}, \overline{r_1}\}$	1
$\{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{r_0, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	9
$\{r_0, \overline{r_1}\}, \{\overline{r_0}, r_1\}, \{\overline{r_0}, r_1\}$	2
$\{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{r_0, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	9
$\{\overline{r_0}, r_1\}, \{\overline{r_0}, \overline{r_1}\}, \{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}$	3

Table 1. Generated Input Sequences

sequences. Notice that the output sequences are lasso-shaped; therefore, we could extend each test sequence indefinitely. However, each test must be finite, so we repeat the recurrent part of the lasso-shaped sequence only once.

In order to evaluate the effectiveness of the generated test suite, we used Strix [8] to synthesize a Mealy Machine from the same specification and we run the generated tests on it. A representation of the synthesized transition system is depicted in Figure 3. Out of the 30 test cases, 23 of them were successful, i.e., the output sequence produced by the transition system was in the set of output sequences generated by the algorithm, while 7 tests failed. Analyzing the failed tests, we observe two phenomena. First, let's consider the case with input sequence $\sigma_1^i = [\{r_0, \overline{r_1}\}, \{\overline{r_0}, r_1\}, \{r_0, \overline{r_1}\}, \{r_0, \overline{r_1}\}]$. The only generated output is $\sigma_1^o = [\{\overline{g_0}, \overline{g_1}\}, \{g_0, \overline{g_1}\}, \{\overline{g_0}, g_1\}, \{\overline{g_0}, \overline{g_1}\}]$ (where g_0 and g_1 stand for *grant*_0 and *grant*_1, respectively). On the other and, if we run σ_1^i on the transition system in Figure 3, we see that it visits states *S*0, *S*1, *S*5, *S*7 and *S*4, and produces the output sequence $\hat{\sigma}_1^0 = [\{\overline{g_0}, \overline{g_1}\}, \{g_0, \overline{g_1}\}, \{\overline{g_0}, g_1\}, \{g_0, \overline{g_1}\}]$. $\hat{\sigma}_1^0$ is a perfectly valid output sequence, recognized by the language, and the system behaves as we expect. In this case, the algorithm produced a too narrow set of output sequences and failed to find the implemented one. This is due to the lasso-shaped condition and the limit on the sequence length, as explained at the end of Section 3. A less restrictive condition or a longer input sequence would have allowed the algorithm to find the expected output. Now, let's consider the input sequence $\sigma_2^i = [\{r_0, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{\overline{r_0}, \overline{r_1}\}, \{r_0, r_1\}]$. The expected output is $\sigma_2^o = [\{g_0, \overline{g_1}\}, \{g_0, g_1\}, \{g_0, \overline{g_1}\}, \{g_0, g_1\}]$. Running σ_2^i on the synthesized system, the system visits states *S*0, *S*2, *S*7, *S*1 and *S*4, and it outputs $\hat{\sigma}_2^o = [\{g_0, g_1\}, \{g_0, g_1\}, \{g_0, g_1\}, \{g_0, g_1\}]$. In this case we see that the system has a strange behavior, because it keeps g_0 active for two steps even if it is not needed. Although the specification does not forbid this behavior, and $\hat{\sigma}_2^o$ is a perfectly valid sequence, one may prefer to observe output σ_2^o instead. In situations like this, having small sets of output sequences can be beneficial in identifying these subtle behaviors.

Finally, we report that the successful test cases were enough to cover all the states of the synthesized transition systems and 87% of its transitions.

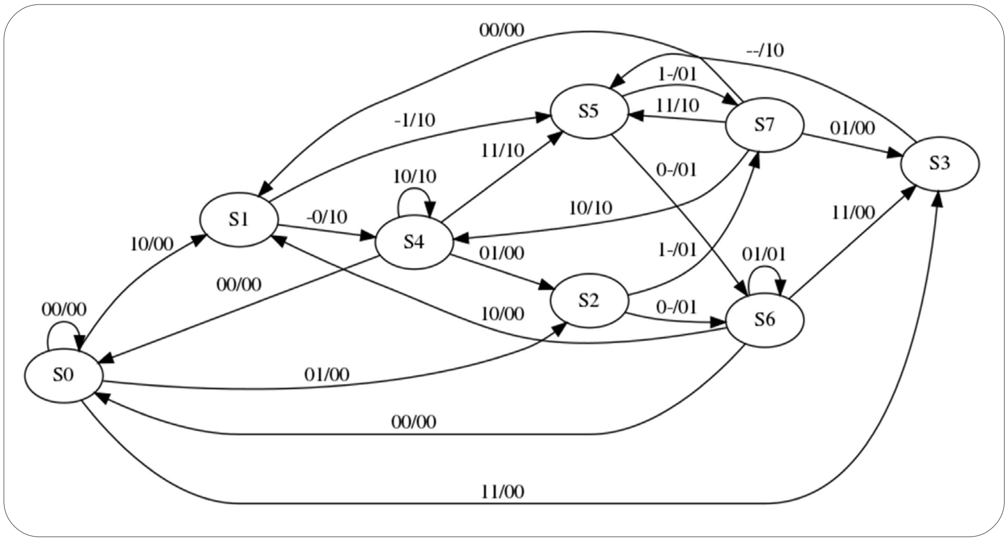

Figure 3. The transition system generated from the full arbiter specification. *S*0 is the initial state and the label on every edge represents the value of the input and output variables. In particular, before the "/" are the values of the *request*_0 and *request*_1 input variables, respectively; after the "/" are the values of corresponding output variables *grant*_0 and *grant*_1, respectively. The "-" symbol means "don't-care", namely it can assume both values

5. Conclusion and Future Work

In this paper we presented an extension of our previous work [12] on automata based test generation. In particular, we propose a new framework that splits the test case generation problem in two parts: first all input sequences are computed, i.e., sequences containing only input atomic propositions, and then the corresponding set of output sequences is selected from the automaton. We presented an algorithm to carry on both these tasks, and we shown its effectiveness using the full arbiter specification in out experimental evaluation. The results reported in Section 4 gave interesting insights into both the challenges as well as the potentials of such approach. In particular, further work is necessary in order to explore and evaluate different generation strategies and outputs selection conditions. Concerning current and future work, our next steps will focus on (1) performing an extensive evaluation of the algorithm with different benchmarks, (2) extending the current idea with new heuristics and selection conditions, and (3) evaluate the possibility to replace the set of output sequences with a more general oracle for the output validation. Finally, SPECPRO is still under active development and we aim at adding new functionalities and exploring more expressive logics.

References

[1] Baier, C., Katoen, J. P. (2008). Principles of model checking. MIT press.

[2] Broy, M., Jonsson, B., Katoen, J. P., Leucker, M., Pretschner, A. (2005). Model-based testing of reactive systems. *In*: Volume 3472 of Springer LNCS. Springer.

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

[4] Faymonville, P., Finkbeiner, B., Tentrup, L. (2017). Bosy: An experimentation framework for bounded synthesis. *In*: Proceedings of CAV. LNCS, vol. 10427, 325-332. Springer.

[5] Filiot, E., Jin, N., Raskin, J. F. (2013). Exploiting structure in ltl synthesis. *International Journal on Software Tools for Technology Transfer* 15 (5-6) 541-561.

[6] Fraser, G., Wotawa, F., Ammann, P. E. (2009). Testing with model checkers: a survey. Software Testing, Verification and Reliability 19 (3) 215-261.

[7] Jacobs, S., Klein, F., Schirmer, S. (2016). A high-level ltl synthesis format: Tlsf v1. 1. arXiv preprint arXiv:1604.02284.

[8] Meyer, P. J., Sickert, S., Luttenberger, M. (2018). Strix: Explicit reactive synthesis strikes back! *In*: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. p. 578-586.

[9] Narizzano, M., Pulina, L., Tacchella, A., Vuotto, S. (2018). 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. 10811, 383-398. Springer.

[10] Narizzano, M., Pulina, L., Tacchella, A., Vuotto, S. (2019). Property specification patterns at work: verification and inconsistency explanation. *Innovations in Systems and Software Engineering*, p. 1-17.

[11] Pnueli, A., Manna, Z. (1992). The temporal logic of reactive and concurrent systems. Springer 16, 12.

[12] Vuotto, S., Narizzano, M., Pulina, L., Tacchella, A. (2019). Automata based test generation with specpro. *In*: Proceedings of the 6th International Workshop on Requirements Engineering and Testing. 13-16. IEEE Press.

[13] Zeng, B., Tan, L. (2016). Test reactive systems with bnuchi-automaton-based temporal requirements. *In*: Theoretical Information Reuse and Integration, 31-57. Springer.