Generating Test Cases for Infinite System Specifications

Stefan Heymer and Jens Grabowski

University of Lübeck, Institute for Telematics, Ratzeburger Allee 160, D-23538 Lübeck, Germany, e–mail: {heymer, grabowsk}@itm.mu-luebeck.de, http://www.itm.mu-luebeck.de

Abstract. Test case generation is a means to validate the implementation of a system *a posteriori* with respect to some given requirements imposed on the system. Current methods for the generation of test cases often rely on the system specification being given as a finite automaton, which does not fully cover the situation in systems with asynchronous communication, In this paper we present an algorithm for the computation of test cases for infinite system specifications.

1 Motivation

Real communication systems are often defined by means of infinite specifications. Reasons for the infiniteness are special assumptions and the characteristics of the used specification languages. For example, the *Specification and Description Language* (SDL) of the ITU-T [1] assumes infinite message buffers for communication purposes and provides abstract data types (ADTs) for the specification of data. The ADT feature allows to specify infinite data ranges.

In most cases conformance between a system specification and a corresponding implementation is checked by means of testing. In the area of communication protocols this type of testing is called *Protocol Conformance Testing* [5]. Conformance testing is based on a set of requirements, the so-called *test purposes*, each of which has to be checked during the procedure of testing. Test purposes are identified, specified and later on implemented (in form of test cases) by hand.

The aim of this paper is to provide a theoretical foundation for proving requirements in a system specification and algorithms for checking these requirements in a corresponding system specification. The latter one can be considered to be work on automatic test generation. For building up a theoretical foundation and algorithms we need suitable formal models for expressing requirements and system specifications.

We restrict ourselves to the treatment of requirements which can be transformed into finite automata [4], i.e., for each requirement we assume there exists an automaton which accepts or generates all traces which exhibit the required property. As formal model for a system specification we use *infinite automata* which is the most general model. Infinite automata and Turing machines accept the same class of languages. By means of a simulator for a given specification it is possible to partially generate the corresponding infinite automaton and to examine its behaviour.

The remainder of this paper is organised in the following manner: Section 2 first gives an account on the theoretical models and equivalences used in our work, while Section 3 describes the generation of test cases for infinite specifications in a step by step manner. We close this paper with some conclusions and directions for further work in Section 4.

2 Some Preliminaries

In this section we give definitions for the formal model underlying our constructions, namely *infinite state automata*, introduce some operations on infinite state automata and discuss equivalence and preorder relations. Also we refer to a standard testing framework used inside the formal methods community. We close this section by taking a look at how to formalise systems and test purposes.

2.1 Infinite Automata and Operations on them

Usually for test case generation finite automata [4] are used. Yet, these are not the right model for our studies, since we are considering *infinite* systems. On the other hand the model of labelled transition systems [7] known from the field of process algebra has its nice points: A number of equivalences are defined on this model formalising different aspects, under which the observable behaviour of two systems can be considered to be equal. Yet, labelled transition systems are not supported with a "built–in" notion of termination. This has to be formalised by special actions.

We will use the model of infinite automata, which seems to embody the "best of both worlds" — the possibly infinite state space of labelled transition systems and the equivalences and preorders defined on them, together with the information on termination from finite automata. An infinite automaton is defined in the following way:

Definition 1 (Infinite State Automaton). An *infinite state automaton* is a tuple

$$\mathcal{A} = \langle \Sigma, \Lambda, \delta, \iota, \surd \rangle \,,$$

where Σ is a (possibly infinite) set of *states*, Λ is an (possibly infinite) alphabet, $\delta \subseteq (\Sigma \times \Lambda) \times \Sigma$ is the *direct transition relation* of $\mathcal{A}$, $\iota \in \Sigma$ is the *initial state* of $\mathcal{A}$, and $\surd \subseteq \Sigma$ is the set of *final states* of $\mathcal{A}$.

In the following, we will write $q \xrightarrow{a} q'$ instead of $((q, a), q') \in \delta$ and $q\surd$ instead of $q \in \surd$. □

In this formulation the alphabet Λ of an infinite automaton $\mathcal{A}$ is meant to be the generator of a monoid whose operation is concatenation of strings — hence, Λ also has to contain an neutral element **1** with respect to this operation that corresponds to the special action τ denoting silent transitions known from labelled transition systems.

This definition gives some nice properties to infinite automata: If one takes a look at suitably defined categories of finite automata, infinite automata and labelled transition systems (as it has been shown for labelled transition systems in [7]), one easily finds injection functors from the categories of finite automata and labelled transition systems into the category of infinite automata. This allows us to "lift" definitions given on labelled transition systems to the realm of infinite automata.

Using this strong correspondence between labelled transition systems and infinite automata, we define operations on infinite automata. As a basic set of operations we take those given in [7], restricting our attention to the product, relabelling and restriction operators defined there.

Definition 2 (Operations on Infinite Automata). Let $\mathcal{A}_1 = \langle \Sigma_1, \Lambda_1, \delta_1, \iota_1, \surd_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma_2, \Lambda_2, \delta_2, \iota_2, \surd_2 \rangle$ be two infinite state automata. We define the following operations:

– $\mathcal{A}_1 \times \mathcal{A}_2 = \langle \Sigma_1 \times \Sigma_2, \Lambda, \delta, (\iota_1, \iota_2), \surd_1 \times \surd_2 \rangle$ is the *product automaton of* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ with the alphabet Λ and the direct transition relation δ being defined as

$$\begin{aligned} \Lambda &= (\Lambda_1 \times \{*\}) \cup (\{*\} \times \Lambda_2) \cup (\Lambda_1 \times \Lambda_2) \\ \delta &= \ \ \{(((q_1, q_2), (a, *)), (q_1', q_2)) \mid ((q_1, a), q_1') \in \delta_1 \wedge q_2 \in \Sigma_2\} \\ &\quad \cup \{(((q_1, q_2), (*, a)), (q_1, q_2')) \mid ((q_2, a), q_2') \in \delta_2 \wedge q_1 \in \Sigma_1\} \\ &\quad \cup \{(((q_1, q_2), (a_1, a_2)), (q_1', q_2')) \mid ((q_1, a_1), q_1') \in \delta_1 \wedge ((q_2, a_2), q_2') \in \delta_2\}, \end{aligned}$$

where $*$ denotes a special *idling* transition.

– Let $\Lambda \subseteq \Lambda_1$. Then $\mathcal{A}_1 \upharpoonright \Lambda = \langle \Sigma_1, \Lambda_1 \cap \Lambda, \delta, \iota_1, \surd_1 \rangle$ is the *restriction of* $\mathcal{A}_1$ *to* Λ with the transition relation δ being defined as

$$\delta = \{((q, a), q') \in \delta_1 \mid a \in \Lambda\}.$$

– Let $\Xi : \Lambda_1 \to \Lambda$ be a function being strict on **1**, i. e. with $\Xi(\mathbf{1}) = \mathbf{1}$, for some alphabet Λ. Then $\mathcal{A}_1\{\Xi\} = \langle \Sigma_1, \Lambda, \delta, \iota_1, \surd_1 \rangle$ is the *relabelling of* $\mathcal{A}_1$ *according to* Ξ with the transition relation δ being defined as

$$\delta = \{(((q, \Xi(a)), q') \in (\Sigma_1 \times \Lambda) \times \Sigma_1 \mid ((q, a), q') \in \delta_1\}.$$

□

These three operations serve to define a vast number of parallel composition operators on infinite automata. While the product operator $\times$ formulates a very liberal parallel composition, where each automaton is allowed to make a move at any time, even simultaneously, the restriction operator can be used to cut down such a liberal composition to a set of events that are *allowed* to happen simultaneously. After restricting the set of possible transitions this way, a following relabelling may rename pairs of names of events to the usual event names.

So for instance the LOTOS–style synchronisation operator $||_A$ for synchronisation on events in the set A can be given as

$$\mathcal{A}_1 \,||_A\, \mathcal{A}_2 = ((\mathcal{A}_1 \times \mathcal{A}_2) \restriction \Lambda)\{\Xi\},$$

where Λ and Ξ are defined as

$$\Lambda = ((A_1 \setminus A) \times \{*\}) \cup (\{*\} \times (A_2 \setminus A)) \cup \{(a,a) \mid a \in A\}$$

and

$$\Xi : \begin{cases} (a,*) \mapsto a,\ a \in A_1 \setminus A, \\ (*,a) \mapsto a,\ a \in A_2 \setminus A, \\ (a,a) \mapsto a,\ a \in A. \end{cases}$$

When the synchronisation set A comprises the union of the alphabets of $\mathcal{A}_1$ and $\mathcal{A}_2$, we will use the shorthand notation $\mathcal{A}_1 \,||\, \mathcal{A}_2$, calling this *full synchronisation*. This synchronisation operator will play an important role in Section 3.2.

2.2 Equivalences and Preorders

After transferring some of the definitions for operations on labelled transition systems from [7] to infinite automata, we will now adapt the definitions for some of the equivalence relations and preorders defined on labelled transition systems. One of these equivalences is *strong bisimilarity.*

Why do we need such equivalences? We are already able to express the equivalence of automata in terms of equality of their languages. For distributed and reactive systems, which have to run for an "infinite" period, this often is impracticable. Furthermore, often one also wants information about branches that do not lead to a final state in the automata, which can be gained from interleaving trace equivalence, or about the internal mechanisms of choice between different alternatives in situations of nondeterminism, which can be retrieved from strong bisimilarity.

We will only give a formal definition of the notion of strong bisimilarity (as it was defined in [6] for labelled transition systems) on infinite automata. The formal definitions of interleaving trace equivalence $=_{\text{it}}$, interleaving trace preorder $\leq_{\text{it}}$ and simulation preorder $\leq_{\text{sim}}$ can be obtained in a similar manner from the definitions of their counterparts defined on labelled transitions.

Definition 3 (Strong Bisimilarity). Let $\mathcal{A}_i = \langle \Sigma_i, A_i, \delta_i, \iota_i, \surd_i \rangle$, $i \in \{1,2\}$ be infinite automata. A strong bisimulation between $\mathcal{A}_1$ and $\mathcal{A}_2$ is a relation $R \subseteq \Sigma_1 \times \Sigma_2$ such that the following holds:

1. $(\iota_1, \iota_2) \in R$
2. If $(q_1,q_2) \in R$ and $q_1 \xrightarrow{a} q_1'$ for some $a \in A_1$ and $q_1' \in \Sigma_1$, then there exists some $q_2' \in \Sigma_2$ with $q_2 \xrightarrow{a} q_2'$ and $(q_1', q_2') \in R$
3. If $(q_1,q_2) \in R$ and $q_2 \xrightarrow{a} q_2'$ for some $a \in A_2$ and $q_2' \in \Sigma_2$, then there exists some $q_1' \in \Sigma_1$ with $q_1 \xrightarrow{a} q_1'$ and $(q_1', q_2') \in R$
4. If $(q_1,q_2) \in R$, then $q_1\surd_1$ if and only if $q_2\surd_2$

$\mathcal{A}_1$ and $\mathcal{A}_2$ are *strongly bisimilar*, denoted $\mathcal{A}_1 =_{\text{bis}} \mathcal{A}_2$, if there exists a strong bisimulation between $\mathcal{A}_1$ and $\mathcal{A}_2$. □

2.3 A Standard Testing Framework

We will now introduce a framework for testing which is mostly inspired by De Nicola and Hennessy [2]. It is mostly standard for labelled transition systems, though we adapt it to infinite automata in the same way we did in the last section for equivalences and preorders. As the infinite automata contain explicit termination information, it is possible to simplify the approach presented in [2].

What we are going to formalise is a *testing preorder*, one of the better known implementation relations used in formal systems design. The general idea is that users of a given system should be flexible enough to deal with all the specified possible responses of the system. The interaction mechanism in this framework is based synchronisation, hence the setting can be formalised in the synchronisation product

$$\mathcal{U} \parallel \mathcal{S},$$

where $\mathcal{U}$ is the model for the user and $\mathcal{S}$ the model for the system. If the user fails to interact with the system successfully, this can be associated with a possible deadlock in the behaviour of the full system $\mathcal{U} \parallel \mathcal{S}$. Now, systems should be implemented in a fashion such that users never are disappointed — that is, if a user was able to successfully interact with the system *specification*, he should always be able to interact successfully with an *implementation* of the system. To ensure this property, the behaviour of the user is reflected in a number of tests.

For describing tests, finite automata can be used, with the test automaton reaching a final state if the test was performed successfully. To apply a test automaton $\mathcal{T}$, it is placed in full synchronisation with the system S, i. e.

$$\mathcal{T} \parallel \mathcal{S}.$$

Now, we are able to formulate boolean evaluators defined over infinite automata, of which the most important is *must–testing*:

$$\begin{aligned}&\mathcal{T}\ \textbf{must succeed}\\ &\quad\Longleftrightarrow \forall w \in \Lambda_{\mathcal{T}}^{*}, q \in \Sigma_{\mathcal{T}}.(\iota_{\mathcal{T}} \overset{w}{\Rightarrow} q \Longrightarrow (q\surd_{\mathcal{T}} \vee \exists a \in \Lambda_{\mathcal{T}}, q' \in \Sigma_{\mathcal{T}}.q \overset{a}{\rightarrow} q'))\end{aligned}$$

$$\mathcal{S}\ \textbf{must do}\ \mathcal{T} \Longleftrightarrow (\mathcal{S} \parallel \mathcal{T})\ \textbf{must succeed}\ .$$

The evaluator **must succeed** states that a test application is successful if it cannot get into a visible deadlock (i. e., is not able to perform further transitions) without signalling successful performance of the test. **must do** is a two–place version of **must succeed**.

This interpretation differs from the interpretation of “must–testing” in [2], especially with respect to infinite behaviour. Infinite paths are considered successful in [2] if and only if some finite prefix of the behaviour is successful. In our setup, we consider finite traces in the first place. Infinite traces of visible events not leading to a final state are not taken into account, whereas infinite paths of *invisible* events correspond to *divergence* and are ignored as long as from every state along the path there is an outgoing visible transition. Thus, as a consequence the effect of infinite visible paths is different from that of divergence. Due to the ability to construct arbitrary tests, all visible finite behaviour can be investigated, which as usual is taken to characterise infinite behaviour as well.

In addition to **must succeed**, we define the following evaluator, which is also inspired by [2]:

$$\mathcal{T}\ \textbf{may succeed} \Longleftrightarrow \exists w \in \Lambda_{\mathcal{T}}^{*}, q \in \Sigma_{\mathcal{T}}.\iota_{\mathcal{T}} \overset{w}{\Longrightarrow} q \wedge q\surd_{\mathcal{T}}$$

$$\mathcal{S}\ \textbf{may do}\ \mathcal{T} \Longleftrightarrow (\mathcal{S} \parallel \mathcal{T})\ \textbf{may succeed}\ .$$

This evaluator tests whether there is a successful trace in $\mathcal{T}$ at all. In practice, **may succeed** can be used to test for traces, **may do** again being a two–place variant of **may succeed**.

2.4 Formalising Systems and Test Purposes

In practical testing, the goal is to check certain properties of systems and not to prove behavioural equivalences between the specification and some given implementation. Thus, the results obtained from testing are weaker than that obtainable from verification of behavioural equivalences. Yet, verification quickly gets hard and cumbersome as soon as the size of specification and implementation increase. Hence, behavioural equivalences have a better place in systematic formal development of systems, where the design of implementations is being done *with respect to* equivalence. *A posteriori* testing of an implementation with respect to a specification using behavioural equivalences quickly gets out of hand for realistic systems.

Which assumptions do we make with respect to the system? We have two main points:

– The system has to be representable in terms of infinite automata.
– The system has a known alphabet of input and output symbols.

Both points are reasonable: The semantics for formal description techniques usually can be given in terms of labelled transition systems, which can be transformed into infinite automata, as was mentioned in Section 2.1, and specifications provide for a known alphabet. Furthermore, we require the infinite automaton which models the system to be only *finitely branching.*

The requirements to be tested have to be formalised, too. Here, we assume that only behaviours are of interest that can be expressed in terms of regular languages. Thus, for each requirement we get a so called *test purpose*, a finite automaton $\mathcal{A}$ accepting the language describing the requirement to be tested. This automaton then will be extended to another finite automaton $\mathcal{A}'$ by adding "loop transitions" to the transitions of $\mathcal{A}$, as is being done inside the SaMsTaG method and tool [3], restricting the interactions of a process to just those described by the test purpose as long as the process actively participates in the behaviour of the test purpose.

3 Generating Test Cases

After providing the preliminaries, we are now able to formalise the generation of test cases for infinite systems. Given the model of a system as a labelled transition system embedded into an infinite automaton as described in Section 2.4 and a test purpose formalised as an infinite automaton, we will first give an account on test cases in the context of the testing framework presented in Section 2.3. Then we will show how to construct test cases in a step by step manner.

3.1 What are we looking for?

Having defined a system and a test purpose, what are we looking for? We want to know whether the system shows the property described by the test purpose or not. This can be expressed formally in the proposition

$$\mathcal{S}\,\textbf{may do}\,\mathcal{P},$$

where $\mathcal{S}$ is the model of the system (in terms of infinite automata), $\mathcal{P}$ is the embedding of the test purpose in an infinite automaton, and the evaluator **may do** is defined as in Section 2.3. Informally this sentence states that we are interested to prove that $\mathcal{S}$ and $\mathcal{P}$ have at least one trace, i. e., one sequence of transitions, in common. This is all we are able to show doing tests in a finite amount of time, as must–testing would require us to prove that *all* of the possibly infinitely many traces of $\mathcal{P}$ also can be performed by $\mathcal{S}$.

Yet, though the formal sentence above exactly captures the question we would like to ask, it provides no means to give an answer to this question when asked with respect to the *implementation*, as we cannot require to have a formal model for the implementation. What is the problem?

The problem lies in the full synchronisation between $\mathcal{S}$ and $\mathcal{P}$ used in the definition of **may do**. Depicted in the form of a box diagram, we have the situation shown in Fig. 1. The two subsystems

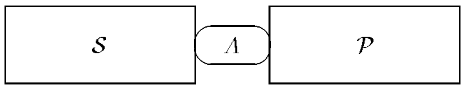

Fig. 1. Full synchronisation between system specification and test purpose

$\mathcal{S}$ and $\mathcal{P}$ communicate and synchronise on the *full* alphabet A of $\mathcal{S}$ and $\mathcal{P}$. Yet, in practical contexts we only are able to observe a subset $I \subseteq A$ of the alphabet of $\mathcal{S}$ and $\mathcal{P}$, that captures the *visible actions* of $\mathcal{S}$ shown at the interface to the environment. But we cannot simply restrict the synchronisation shown in Fig. 1. Hence, we have to find a new subsystem $\mathcal{T}$, as it is shown in Fig. 2. Which properties does the testing subsystem $\mathcal{T}$ have to have? Intuitively, it should act as a kind of "man in the middle" between $\mathcal{S}$ and $\mathcal{P}$, as depicted in Fig. 3. While $\mathcal{S}$ and $\mathcal{P}$ still communicate and synchronise over the full alphabet A, the test process $\mathcal{T}$ communicates with $\mathcal{S}$ *and* $\mathcal{P}$, yet it only synchronises with them over the interface alphabet $\mathcal{I}$.

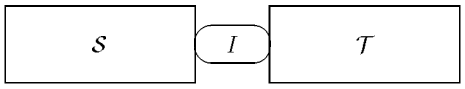

Fig. 2. Partial synchronisation between system specification and test automaton

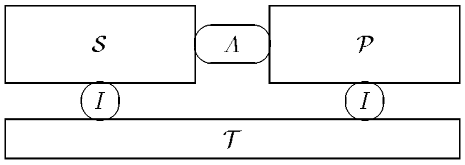

Fig. 3. Inclusion of the test automaton T

Now we can formalise the requirements on $\mathcal{T}$. Firstly, we have

$$\mathcal{S}\,\mathbf{may\ do}\,\mathcal{P}, \tag{1}$$

as already mentioned above. Secondly, as we want to get to reproducible results, we also require

$$\mathcal{P}\,||_I\,\mathcal{T}\ \mathbf{must\ succeed}, \tag{2}$$

such that every trace of $\mathcal{T}$ leading to a *pass* result has to be a trace of $\mathcal{P}$ leading to the same result. Thirdly, we require

$$\mathcal{S}\,||_I\,\mathcal{T}\ \mathbf{must\ succeed}, \tag{3}$$

hence we want to identify the trace in $\mathcal{S}$ corresponding to a trace leading to a *pass* result in $\mathcal{P}$ in a reproducible manner. These last two properties and the intuition from Fig. 3 can be summarised in the proposition

$$\mathcal{T}\,||_I\,(\mathcal{S}\,||_A\,\mathcal{P})\ \mathbf{must\ succeed}\,. \tag{4}$$

In this proposition, the term $\mathcal{S}||_A\mathcal{P}$ comes from expanding the definition of **may do** in proposition (1).

Having found such a tester process $\mathcal{T}$, we may drop the test purpose $\mathcal{P}$ (and thus transit from the configuration of Fig. 3 to the one shown in Fig. 2) from our investigations — as $\mathcal{P}\,||_I\,\mathcal{T}$ **must succeed**, $\mathcal{T}$ will force the right paths to be taken in $\mathcal{P}$, and as $\mathcal{S}\,||_I\,\mathcal{T}$ **must succeed**, this path will be taken in $\mathcal{S}$, too.

So far, we have formulated the solution to our problem inside the standard testing framework. We also place a further requirement on $\mathcal{T}$: It has to be *deterministic*, that is

$$\forall q \in \Sigma_{\mathcal{T}}, a \in A_{\mathcal{T}}.\exists q', q'' \in \Sigma_{\mathcal{T}}.$$

This requirement comes from practical considerations: when performing a test on an implementation, we cannot rely on angelic choices, i. e. that decisions in case of nondeterminism being made only "the right way". This way, a test can be performed without having to backtrack to earlier points, where a wrong decision has been made.

3.2 Building Test Cases

Now that we know what we are looking for, how are we going to achieve it? Our test purpose $\mathcal{P}$ is finite and exhibits only a finite behaviour, yet our system specification may have an infinite number of states.

Hence we will have to resort to some tricks to make the system specification manageable. We know quite well that the *unfolding* of a labelled transition system is observably equal to the original transition system in a very strong sense. As we have the very strong connection between infinite automata and labelled transition systems mentioned in Section 2.1, we are able to lift this result for use with infinite automata. We first define the notion of an unfolding of an infinite automaton.

Definition 4. Let $\mathcal{A} = \langle \Sigma, \Lambda, \delta, \iota, \surd \rangle$ be an infinite automaton. Define a sequence of infinite automata $(\mathcal{A}_i)_{i \in \mathbb{N}}$ such that for $\mathcal{A}_i = \langle \Sigma_i, \Lambda, \delta_i, (\iota, *), \surd_i \rangle$ with $i \in \mathbb{N}$ the state space is being inductively defined as

$$\begin{aligned} \Sigma_1 &= \{(\iota, *)\} \\ \Sigma_{i+1} &= \{(q', a')(q, a)\sigma \in (\Sigma \times \Lambda)^* \mid (q, a)\sigma \in \Sigma_i \land q \xrightarrow{a'} q'\}, \end{aligned}$$

the transition relation $\delta_i \subseteq (\Sigma_i \times \Lambda) \times \Sigma_i$ is being given by

$$(q, a)\sigma \xrightarrow{a'}_i (q', a')(q, a)\sigma \iff q \xrightarrow{a'} q',$$

and the termination predicate is being given as

$$\surd_i = \{(q, a)\sigma \in \Sigma_i \mid q\surd\}.$$

The *unfolding of* $\mathcal{A}$ then is defined as

$$\text{unfolding } \mathcal{A} = \left\langle \bigcup_{i=1}^{\infty} \Sigma_i, \Lambda, \bigcup_{i=1}^{\infty} \delta_i, (\iota, *), \bigcup_{i=1}^{\infty} \surd_i \right\rangle.$$

□

Defining the unfolding of an infinite automaton this way, we find the following strong equivalence between an infinite automaton and its unfolding.

Theorem 5. *Let $\mathcal{A} = \langle \Sigma, \Lambda, \delta, \iota, \surd \rangle$ be an infinite automaton. Then the following proposition holds:*

$$\mathcal{A} =_{\text{bis}} \text{unfold } \mathcal{A}$$

Proof. Follows directly from the construction of the unfolding. □

Abusing mathematical notation a bit, we see that $\lim_{n \to \infty} \mathcal{A}_i =_{\text{bis}} \mathcal{A}$ holds, where $\mathcal{A}_i$ are the elements of the sequence defined in Definition 4. Moreover, if $\mathcal{A}$ is only finitely branching, we observe that each of the approximating automata $\mathcal{A}_i$ is finite. Moreover, we get the following result:

Theorem 6. *Let $\mathcal{A}$ be an infinite automaton, and let $(\mathcal{A}_i)_{i \in \mathbb{N}}$ be a sequence of infinite automata constructed as shown in Theorem 5. Then*

$$\forall i \in \mathbb{N}. \mathcal{A}_i \leq_{\text{sim}} \mathcal{A}$$

holds.

Proof. Straightforward. □

This enables us to build test cases in an approximation process. We will start by unfolding the model of the specification, and iteratively apply the steps described in the following paragraphs. In a first step we compute the approximation of the specification $\mathcal{S}_i$ for some depth i by defining

$$\mathcal{S}_i = \langle \Sigma_i, \Lambda, \delta \cap (\Sigma_i \times \Lambda) \times \Sigma_i, (\iota, *), \surd_i \cap \Sigma_i \rangle,$$

where the state space Σ_i is given as

$$\Sigma_i = \{\sigma \in \Sigma_{\text{unfold } \mathcal{S}} \mid \#_I(\sigma) \leq i\},$$

with the function $\#_I$ being inductively defined as

$$\begin{aligned} \#_I(\epsilon) &= 0 \\ \#_I((q, a)\sigma) &= \begin{cases} 1 + \#_I(\sigma) & a \in I \\ \#_I(\sigma) & \text{otherwise} \end{cases} \end{aligned}$$

counting occurrences of actions observable at the interface.

Then, we fully synchronise this approximation with the test purpose $\mathcal{P}$. This yields an automaton

$$\mathcal{S}_i \,||_\Lambda\, \mathcal{P}.$$

We now clearly have $\mathcal{S}_i \,||_\Lambda\, \mathcal{P} \leq_{\text{sim}} \mathcal{S} \,||_\Lambda\, \mathcal{P}$, and hence

$$\mathcal{S}_i \,||_\Lambda\, \mathcal{P} \textbf{ must succeed} \Longrightarrow \mathcal{S} \,||_\Lambda\, \mathcal{P} \textbf{ may succeed} .$$

By virtue of mathematical logic this proposition holds even when $\mathcal{S}_i \,||_\Lambda\, \mathcal{P}$ **may succeed** does not hold. But, for the following considerations we assume that $\mathcal{S}_i \,||_\Lambda\, \mathcal{P}$ **may succeed** holds.

For sake of notational simplicity, we will define $\mathcal{A}_i = \mathcal{S}_i \,||_\Lambda\, \mathcal{P}$. Hence we now have a tree–shaped finite automaton $\mathcal{A}_i$, that can even be used as testing automaton, yielding

$$\mathcal{S} \textbf{ may do } \mathcal{A}_i \iff \mathcal{S} \,||_\Lambda\, (\mathcal{S}_i \,||_\Lambda\, \mathcal{P}) \textbf{ may succeed} .$$

Yet, $\mathcal{A}_i$ still contains too much information, as it has to be synchronised with the specification over the full alphabet Λ. Hence in the next step we will abstract from all actions in $\mathcal{A}_i$ that are not observable from the interface between the test automaton and the model of the specification. Thus we define the automaton

$$\mathcal{A}_i' = \mathcal{A}_i\{\delta_I\},$$

where the abstraction function δ_I is defined as

$$\delta_I(a) = \begin{cases} a, \text{ if } a \in I \\ \mathbf{1}, \text{ otherwise} \end{cases}$$

with I being the alphabet of the interface between the specification and its environment and $\mathbf{1}$ being the monoid unit. Again, we have

$$\mathcal{S} \,||_I\, \mathcal{A}_i' \textbf{ may succeed} .$$

In the next step, we now will drop all the silent actions $\mathbf{1}$ from the automaton. For this, we first need an auxiliary definition.

Definition 7. Let $\mathcal{A} = \langle \Sigma, \Lambda, \delta, \iota, \surd \rangle$ be an infinite automaton. Define the $\mathbf{1}$*-closure* of $\mathcal{A}$ to be the automaton closure $\mathcal{A} = \langle \Sigma, \Lambda, \Delta_\mathcal{A}, \iota, \surd \rangle$ with the transition relation $\Delta_\mathcal{A}$ being given as

$$\Delta_\mathcal{A} = \{((s,a),s') \in (\Sigma \times \Lambda) \times \Sigma \mid \exists s_1, s_2 \in \Sigma . s \xrightarrow{\mathbf{1}}^* s_1 \xrightarrow{a} s_2 \xrightarrow{\mathbf{1}}^* s'\},$$

where $\xrightarrow{\mathbf{1}}^*$ is the transitive and reflexive closure of $\xrightarrow{\mathbf{1}}$. □

So now we are able to define a third automaton A_i'' as

$$\mathcal{A}_i'' = \text{closure } \mathcal{A}_i'.$$

For this automaton, we immediately observe that

$$\mathcal{A}_i'' =_{\text{it}} \mathcal{A}_i'$$

holds, as the monoid unit $\mathbf{1}$ does not contribute to traces in the automaton. We still have

$$\mathcal{S} \,||_I\, \mathcal{A}_i'' \textbf{ may succeed},$$

although the moment of choice between alternative paths in $\mathcal{A}_i''$ has been changed with respect to $\mathcal{A}_i$.

So now we have a automaton $\mathcal{A}_i''$ that only contains actions visible at the interface between the specification automaton $\mathcal{S}$ and its environment, and that is not capable of spontaneous transitions. This nearly is what we need — the only point missing is that $\mathcal{A}_i''$ is not deterministic. This will be fixed in our next step, but first we have again to give an auxiliary definition.

Definition 8. Let $\mathcal{A} = \langle \Sigma, A, \delta, \iota, \surd \rangle$ be an infinite automaton. A sequence $(\mathcal{B}_i)_{i \in \mathbb{N}}$ of infinite automata is defined as follows:

- $\mathcal{B}_1 = \langle \{\{\iota\}\}, A, \emptyset, \{\iota\}, \surd_1 \rangle$, where $\surd_1 = \{\{\iota\}\}$ if $\iota \in \surd$, and $\surd_1 = \emptyset$ otherwise.
- $\mathcal{B}_{i+1} = \langle \Sigma_{i+1}, A, \delta_{i+1}, \{\iota\}, \surd_{i+1} \rangle$, where the transition relation δ_{i+1} is given as

$$\delta_{i+1} = \{((S,a),S') \mid S \in \Sigma_i \wedge a \in A \wedge S' = \{s \in \Sigma \mid \exists s' \in S . s \xrightarrow{a} s'\}\},$$

the state space Σ_{i+1} is given as

$$\Sigma_{i+1} = \{S \mid (S,a,S') \in \rightarrow_{i+1}\} \cup \{S' \mid ((S,a),S') \in \delta_{i+1}\},$$

and the termination predicate $\surd_{i+1}$ is given as

$$\surd_{i+1} = \{S \in \Sigma_{i+1} \mid \exists s \in S . s \surd\}.$$

We define the *determinisation of* $\mathcal{A}$ (denoted $\det \mathcal{A}$) to be

$$\det \mathcal{A} = \bigcup_{i=1}^{\infty} \mathcal{B}_i .$$

□

The process of determinisation terminates for finite automata, hence we are able to define

$$\mathcal{A}_i''' = \det \mathcal{A}_i''$$

without loosing termination of our test generation process. So now we have a deterministic automaton without spontaneous transitions, which is labelled just with the actions visible at the interface between the specification and its environment. For this automaton again

$$\mathcal{A}_i''' =_{\mathrm{it}} \mathcal{A}_i''$$

holds, and again we have

$$\mathcal{S} \parallel_I \mathcal{A}_i''' \ \textbf{may succeed} \ .$$

Yet, we are not quite at the point we wanted to get to, as still only $\mathcal{S} \parallel_I \mathcal{A}_i'''$ **may succeed** holds instead of $\mathcal{S} \parallel_I \mathcal{A}_i'''$ **must succeed**, as would have to be the case if A_i''' would be a test automaton. Due to the construction used in the process of determinisation, the final states of $\mathcal{A}_i'''$ contain also states that lead back to states of the specification $\mathcal{S}$[1] which themselves are *no* final states of $\mathcal{S}$. We have to fix this by determining a sub-automaton of $\mathcal{A}_i'''$ in the following way:

$$\mathcal{A}_i'''' = \langle \Sigma_{\mathcal{A}_i'''}, A_{\mathcal{A}_i'''}, \delta_{\mathcal{A}_i'''}, \iota_{\mathcal{A}_i'''}, \surd \rangle ,$$

where the termination predicate $\surd$ is given as

$$\surd = \{S \in \Sigma_{\mathcal{A}_i'''} \mid \forall s \in S . s \surd_{\mathcal{S} \times \mathcal{P}}\},$$

such that the resulting automaton has as its final states only those sets of states from $\mathcal{A}_i'''$ containing *exclusively* final states of the automaton $\mathcal{A}_i''$. This way it is clear that the test purpose $\mathcal{P}$ was successfully performed when a trace is found which leads from the initial state of $\mathcal{A}_i''''$ to one of its final states.

With the construction shown above, we have $\mathcal{A}_i'''' \leq_{\mathrm{it}} \mathcal{A}_i'''$, as we have only less final states in $\mathcal{A}_i''''$, and

$$\mathcal{S} \parallel_I \mathcal{A}_i'''' \ \textbf{may succeed}$$

as expected.

We proposed these steps to form an approximation process. Dropping our assumption stated above that $\mathcal{S}_i \parallel_A \mathcal{P}$ **may succeed**, we are now able to formulate an algorithm. The approximation stops as soon as the termination predicate of $\mathcal{A}_i''''$ is not empty, i. e. a unique final state has been found. This leaves us with the following approximation algorithm:

[1] Note that by construction of the parallel composition the states of $\mathcal{S} \parallel_I \mathcal{A}_i'''$ are pairs of states.

```
1. $i := 0$.
2. Compute $\mathcal{S}_i$.
3. $\mathcal{A}'_i := (\mathcal{S}_i \parallel_\Lambda \mathcal{P})\{\delta_I\}$.
4. Compute $\mathcal{A}''_i$ as described above.
5. $\mathcal{A}'''_i = \det \mathcal{A}''_i$.
6. Compute $\mathcal{A}''''_i$ as described above.
7. Does $\surd_{\mathcal{A}''''_i} \neq \emptyset$ hold? If yes, proceed with Step 9, otherwise go on.
8. Increment $i$ by 1, proceed with Step 2.
9. $\mathcal{T}' := \mathcal{A}''''_i$.
```

In each Step of this algorithm, we have $\mathcal{A}''''_i \leq_{\text{it}} \mathcal{A}'''_i =_{\text{it}} \mathcal{A}'''_i =_{\text{it}} \mathcal{A}'_i$, furthermore we have $\mathcal{S}_i \leq_{\text{sim}} \mathcal{S}$ as stated above. The result of this algorithm is a suitable automaton $\mathcal{T}'$, which only fulfils

$$\mathcal{S} \parallel_I \mathcal{T}' \textbf{ may succeed},$$

as $\mathcal{T}'$ may still contain "dead" branches leading to leaves that are no final states. So we still have to prune these dead branches. Hence we define the testing automaton to be

$$\mathcal{T} = \langle \Sigma_\mathcal{T}, \Lambda_{\mathcal{T}'}, \delta_\mathcal{T}, \iota_{\mathcal{T}'}, \surd_{\mathcal{T}'} \rangle$$

with the state space $\Sigma_\mathcal{T}$ being cut down to

$$\Sigma_\mathcal{T} = \{ s \in \Sigma_{\mathcal{T}'} \mid \exists s' \in \surd_{\mathcal{T}'}. s \sqsubseteq s' \},$$

where the relation $\sqsubseteq \subseteq \Sigma_{\mathcal{T}'} \times \Sigma_{\mathcal{T}'}$ is defined as the reflexive and transitive closure of the relation

$$\{ (s, s') \in \Sigma_{\mathcal{T}'} \times \Sigma_{\mathcal{T}'} \mid \exists a \in \Lambda. s \xrightarrow{a}_{\mathcal{T}'} s' \}.$$

The transition relation $\delta_{\mathcal{T}'}$ likewise is cut down to

$$\delta_\mathcal{T} = \{ ((s, a), s') \in \delta_{\mathcal{T}'} \mid s \in \Sigma_\mathcal{T} \land s' \in \Sigma_\mathcal{T} \}.$$

With $\mathcal{T}$, we finally have the testing automaton we were looking for, as this automaton fulfils the proposition

$$\mathcal{S} \parallel_I \mathcal{T} \textbf{ must succeed} .$$

4 Conclusions

The work presented in this paper facilitates the automatic computation of test cases for systems that are described by means of infinite automata as well as by means of finite automata. The algorithm described can be varied in some places, so e. g. the approximation of a system can be done with a different sequence of finite automata, for example by doing a k–bounded depth–first search. This way, the work presented here also can be seen as a theoretical basis for the SaMsTaG method and tool [3].

With this said, the work described here is not finished — we plan to extend the approach presented here to consider partially ordered traces, which would reduce the size of test cases drastically. We also plan to formalise tests on models containing a notion of time in a further step.

References

1. ITU Telecommunication Standards Sector SG 10. ITU–T Recommendation Z.100: Specification and Description Language (SDL) (formerly CCITT Recommendation Z.100). ITU, Geneva, June 1992.
2. R. De Nicola and M.C.B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.
3. J. Grabowski, D. Hogrefe, and R. Nahm. Test Case Generation with Test Purpose Specification by MSCs. In O. Færgemand and A. Sarma, editors, *SDL'93 Using Objects*. North-Holland, 1993.
4. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.
5. ISO. *Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework*. International Standard IS-9646. ISO, Geneve, 1991. Also: CCITT X.290–X.294.
6. D. Park. Concurrency and Automata on Infinite Sequences. In P. Deussen, editor, *Proceedings 5th GI Conference*, pages 167–183. Lecture Notes in Computer Science 104, Springer-Verlag, 1981.
7. G. Winskel and M. Nielsen. *The Handbook of Logic in Computer Science*, chapter Models for Concurrency. Springer–Verlag, 1992.