Taming Interface Specifications*

Tiziana Margaria[1], A. Prasad Sistla[2], Bernhard Steffen[3], and Lenore D. Zuck[2]

[1] Georg-August-Universität Göttingen
margaria@informatik.uni-goettingen.de
[2] University of Illinois at Chicago
{sistla,lenore}@cs.uic.edu
[3] Universität Dortmund
Bernhard.Steffen@cs.uni-dortmund.de

Abstract. Software is often being assembled using third-party components where the developers have little knowledge of, and even less control over, the internals of the components comprising the overall system. One obstacle to composing agents is that current formal methods are mainly concerned with "closed" systems that are built from the ground up. Such systems are fully under the control of the user. Hence, problems arising from ill-specified components can be resolved by a close inspection of the systems. When composing systems using "off-the-shelf" components, this is often no longer the case.

The paper addresses the problem of *under-specification*, where an off-the-shelf component does only what it claims to do, however, it claims more behaviors than it actually has and that one wishes for, some of which may render it useless. Given such an under-specified module, we propose a method to automatically synthesize some safety properties from it that would tame its "bad" behaviors. The advantage of restricting to safety properties is that they are monitorable.

The safety properties are derived using an automata-theoretic approach. We show that, when restricting to ω-regular languages, there is no maximal safety property. For this case we construct a sequence of increasingly larger safety properties. We also show how to construct an infinite-state automata that can capture any safety property that is contained in the original specifications.

1 Introduction

The process of constructing software is undergoing rapid changes. Instead of a monolithic software development within an organization, increasingly, software is being assembled using third-party components (e.g., JavaBeans, .NET, etc.). The developers have little knowledge of, and even less control over, the internals of the components comprising the overall system.

One obstacle to composing agents is that current formal methods are mainly concerned with "closed" systems that are built from the ground up. Such systems are fully under the control of the user. Hence, problems arising from ill-specified components can

* This research was supported in part by NSF grants CCR-0205571 and CCR-0205363, and ONR grant N00014-99-1-0131

be resolved by a close inspection of the systems. When composing agents using "off-the-shelf" ones, this is often no longer the case. Out of consideration for proprietary information, or in order to simplify presentation, companies may provide *incomplete specifications*. Worse, some agents may have no description at all except one that can be obtained by experimentation. Despite being ill-specified, "off-the-shelf" components might still be attractive enough so that the designer of a new service may wish to use them. In order to do so safely, the designer must be able to deal with the possibility that these components may exhibit undesired or unanticipated behavior, which could potentially compromise the correctness and security of the new system.

The main problem addressed in this paper is that of *under-specification*. As a simple example of the phenomenon, consider an interface specification that guarantees "after input *query* q is received, output $r = \mathit{response}(q)$ is produced." The designer of the interface probably meant a stronger specification, "after q is received, nothing else is produced until r is produced." Assume that the later version is sufficient and necessary to ensure the correctness of the entire system consisting of the module and the interface. Formal methods in general, and model checking in particular, are to fail in such situations since there is no algorithmic way to provide the model checker with the proper strengthening of the interface specification. Yet, under the assumption that interface specifications may be partial, there may exist a subset of the allowed behaviors that guarantees correctness, and one may still choose to use the component, provided deviations of the interface from this "good" set of behaviors can be detected at runtime.

Assume that we are given

- A finite-state *module* M, designed by our designer and accompanied by the full details of its implementation;
- An *interface specification* Φ_I for the external component interacting with the module M; and
- A *goal specification* Φ for the entire system which must be satisfied by the interaction between the module and the interface.

The *system* thus contains the composition of the module with the external component. The goal of the designer is to guarantee that the behavior of the system satisfies the goal specification Φ. Obviously, our underlying assumption is that the external component is helpful for the module, i.e., it computes things that the module cannot accomplish on its own. For example, if the module is a "general best buyer," and the external component has access to numerous bookstores which the module has no access to, the module uses the component to obtain the best book deals. However, the book buying component may be under-specified, thus, allow for behaviors for which the designer cannot guarantee the goal (while, of course, allow also for "good" behaviors, otherwise the designer will not be inclined to use it!).

The designer has a reason to believe that the real interface specification is more restricted than Φ_I, say it is $\Phi_I \wedge \phi$ for some ϕ. With this assumption, the designer can compose the module with the component so that Φ is guaranteed. If the property ϕ can be *run-time monitored*, i.e., if there is a simple module that runs synchronously with the system and watches for violations of ϕ, the designer can then go ahead and safely use the designed module as long as the monitor does not alarm.

Using a run-time monitor would allow the system to operate correctly as long as the external component satisfies ϕ. When it violates it, the run-time monitor alerts the user of the system that a violation occurred (and $\varPhi$ is no longer guaranteed). However, this should not be viewed as a major obstacle – the applications intended are clearly not "safety critical" since no designer would use "black box" components inside a safety critical application. Such components can only be used in applications where a violation is tolerable. E.g., a leak of the credit card number, if caught in a timely manner, allows the holder of the credit card to alert the credit company and avoid bogus charges.

In this paper we focus on the problem of synthesizing a property ϕ that can be run-time monitored. In fact, we restrict the search to *safety* properties. Safety properties are those that can only be violated by a finite prefix. Hence, they can be monitored. In future work we will show how to synthesize the module M. Here, we restrict to the case where M is trivial. Thus, given $\varPhi_I$, we synthesize a safety property ϕ such that $\varPhi_I \ \wedge \ \phi \rightarrow \varPhi$.

We consider properties that are expressed as ω-sequences over a finite alphabet. Essentially, our synthesis problem reduces to that of finding safety properties that are contained in the property defined by $\varXi = \neg\varPhi_I \vee \varPhi$. While there is always *some* safety property ϕ that guarantees $\varPhi_I \ \wedge \ \phi \rightarrow \varPhi$ (e.g., the trivially false property), there is, in general no "maximal" one: Assume that $\varXi$ is neither valid nor an obvious safety property. We show that when $\varXi$ is ω-regular, then for every safety property ϕ_1 such that $\varPhi_I \wedge \phi_1 \rightarrow \varPhi$ there exists is a safety property $\phi_2 \neq \phi_1$ that is implied by ϕ_1 and that satisfies $\varPhi_I \ \wedge \ \phi_2 \rightarrow \varPhi$.

We compute a family of safety properties ϕ_k such that the higher k is, the more "accurate" and costly is the computation of ϕ_k. All these safety properties are given by deterministic finite state automata. As to be expected, the number of states of these automata increases linearly with k.

We also define a class of, possibly infinite-state, deterministic automata called *bounded automata* and show that the set of sequences accepted by bounded automata gives the desired safety property ϕ. We also prove a completeness result, showing that every safety property contained in the property defined by $\varXi$ is accepted by some bounded automaton. In order for these automata to be useful, they need to be recursive, i.e., computable. With this in mind, we define *history-based recursive automata* that can be applied in practice.

The paper is organized as follows. Section 2 introduces the notation and definitions. Section 3 establishes the impossibility of finding a maximal safety properties for the case of ω-regular languages and contains the construction of the sequence of finite-state ω-automata for the synthesis of the desired safety property. Section 4 contains the definitions and results of our study of bounded automata. Section 5 compares our work with related work, and Section 6 contains discussion and concluding remarks.

2 Preliminaries

Sequences. Let S be a finite set. Let $\sigma = s_0, s_1, \ldots$ be a possibly infinite sequence over S. The length of σ, $|\sigma|$, is defined to be the number of elements in σ if σ is finite, and ω otherwise. We let ; denote the concatenation operator for sequences so that if

α_1 is a finite sequence and α_2 is a either a finite or a ω-sequence then $\alpha_1;\alpha_2$ is the concatenation of the two sequences in that order.

For integers i and j such that $0 \le i \le j < |\sigma|$, $\sigma[i,j]$ denotes the (finite) sequence $s_i, \ldots s_j$. A *prefix* of σ is any $\sigma[0,j]$ for $j < |\sigma|$. We denote the set of σ's prefixes by $Pref(\sigma)$. Given an integer i, $0 \le i < |\sigma|$, we denote by $\sigma^{(i)}$ the *suffix* of σ that starts with s_i.

For an infinite sequence $\sigma : s_0, \ldots$, we denote by $\inf(\sigma)$ the set of S-elements that occur in σ infinitely many times, i.e., $\inf(\sigma) = \{s : s_i = s \text{ for infinitely many } i\text{'s}\}$.

Languages. A *language* L over a finite alphabet Σ is a set of finite or infinite sequences over Σ. When L consists only of infinite strings (sequences), we sometimes refer to it as an ω*-language*. For a language L, we denote the set of prefixes of L by $Pref(L)$, i.e.,

$$Pref(L) = \bigcup_{\sigma \in L} Pref(\sigma)$$

Following [6,2], an ω-language L is a *safety property* if for every $\sigma \in \Sigma^{\infty}$:

$$Pref(\sigma) \subseteq Pref(L) \quad \Longrightarrow \quad \sigma \in L$$

i.e., L is a safety property if it is *limit closed* – for every ω-string σ, if every prefix of σ is a prefix of some L-string, then σ must be an L-string.

Safety properties play an important role in the results reported here.

Büchi Automata A *Büchi automaton* (NBA for short) $\mathcal{A}$ on infinite strings is described by a quintuple $(Q, \Sigma, \delta, q_0, F)$ where:

– Q is a finite set of states;
– Σ is a finite alphabet of symbols;
– $\delta \colon Q \times \Sigma \rightarrow 2^Q$ is a transition function;
– $q_0 \in Q$ is an initial state; and
– $F \subseteq Q$ is a set of accepting states.

The *generalized transition function* $\delta^* : Q \times \Sigma^* \rightarrow 2^Q$ is defined in the usual way, i.e., for every state q, $\delta^*(q, \epsilon) = \{q\}$, and for any $\sigma \in \Sigma^*$ and $a \in \Sigma$, $\delta^*(q, \sigma; a) = \cup_{q' \in \delta^*(q,\sigma)} \delta(q', a)$.

If for every $(q, a) \in Q \times \Sigma$, $|\delta(q, a)| = 1$, then $\mathcal{A}$ is called a *deterministic* Büchi automaton (or DBA for short).

Let $\sigma : a_1, \ldots$ be an infinite sequence over Σ. A *run* r *of* $\mathcal{A}$ on σ is an infinite sequence $q^0, \ldots$ over Q such that:

– $q^0 = q_0$;
– for every $i > 0$, $q^i \in \delta(q^{i-1}, a_i)$;

A run r is *accepting* if $\inf(r) \cap F \neq \emptyset$. The automaton $\mathcal{A}$ *accepts* the ω-string σ if it has an accepting run over σ (for the case of DBAs, the automaton has a single run over σ). The *language accepted by* $\mathcal{A}$, denoted by $L(\mathcal{A})$, is the set of ω-strings that $\mathcal{A}$ accepts. A language L' is called ω*-regular* if it is an ω-language that is accepted by some (possibly non-deterministic) Büchi automaton.

A Büchi automaton $\mathcal{A}$ can also be used to define a *regular automaton* that is just like $\mathcal{A}$, only the acceptance condition of a run r is that its last state is accepting. We denote the regular language accepted by the regular version of $\mathcal{A}$ by $L_f(\mathcal{A})$.

3 Synthesis of Safety Properties by Finite-State Automata

As described in Section 1, given an interface specification Φ_I of a readily available off-the-shelf reactive component and a desired goal specification Φ, we wish to derive a safety property ϕ so that $\Phi_I \;\wedge\; \phi \rightarrow \Phi$. We assume that both Φ_I and Φ are given by temporal logic formulas. Our methods, however, can also be applied to the case where Φ_I and Φ are described by ω-automata. As before, denote $\Xi = \neg\Phi_I \;\vee\; \Phi$. Obviously, any safety property ϕ such that $\phi \rightarrow \Xi$ is a satisfies our requirements.

In this section we describe how to obtain ϕ as deterministic automaton. The advantage of obtaining the required ϕ as a deterministic automaton is that it can be directly used to monitor the execution of the module: The automaton simply runs on the executions of the module and a violation of the safety property by the execution is indicated by the automaton entering a "bad" state. In this section we restrict to ω-automata, for which, as we show, only a limited set of safety properties can be derived. To overcome this limitation we present, in the next section, automata that are not necessarily finite-state and study their power.

Using the methods of [13,3], we first obtain a Büchi automaton $\mathcal{A}$ whose language is the set of ω-strings satisfying Ξ. Thus, we reduce the problem to that of obtaining a deterministic automaton whose language is a safety property that is contained in $L(\mathcal{A})$. Roughly speaking, we start with the automaton that accepts Ξ, and construct a family of automata, indexed by some integer k, each accepting a sequence that satisfies Ξ where an accepting state is realized in every block of k consecutive states.

Example 1. Suppose an off-the-shelf *permission manager* that receives requests by a user and grants appropriate permissions, e.g., authorizations to access different resources. Assume there are two types of requests, r_1 and r_2, with two corresponding grants, g_1 and g_2 respectively. The permission manager guarantees that every request is eventually responded by granting of the corresponding permission. Thus Φ_I is:

$$\Box(r_1 \rightarrow \Diamond\, g_1) \;\wedge\; \Box(r_2 \rightarrow \Diamond\, g_2)$$

Assume a user who wishes to use e component and who requires that a r_1 request receives a higher priority than a r_2 request, at the possibly cost of ignoring an r_2 request, i.e., that an r_1 should be granted before any potentially pending r_2 requests are granted. Thus, the goal Φ of the user is:

$$\Box(r_1 \;\rightarrow\; (\neg g_2)\,\mathcal{U}\, g_1)$$

where $\mathcal{U}$ is the temporal "until" operator.

Note user's requirement for r_1 is stronger than that guaranteed by the component, while the user's requirement for r_2 is weaker.

The user can construct a monitor that monitors for violations of a safety property that is contained in $(\neg\Phi_I \vee \Phi)$, e.g., of the property:

$$\Box(r_1 \;\rightarrow\; (\neg g_2)\,\mathcal{W}\, g_1)$$

where $\mathcal{W}$ is the unless (weak until) temporal operator. Thus, the property does not require g_1 to hold after r_1 (but does require that as long as g_1 doesn't hold, neither does g_2).

3.1 Derivation of a Safety Property Using Büchi Automata

Ideally, given a property described by a Büchi automaton $\mathcal{A}$, we would like to synthesize the *maximal* safety property that is contained in $L(\mathcal{A})$. However, as the following lemma shows, if $L(\mathcal{A})$ is not already a safety property, then there exists no maximal safety property in it that can be accepted by a Büchi automaton. The proof of this lemma is given at the end of this subsection.

Lemma 1. *Let $\mathcal{A}$ be a Büchi automaton and assume $L(\mathcal{A})$ is not a safety property. Then for every safety property $L' \subset L(\mathcal{A})$, there exists a safety property L'' such that $L' \subset L'' \subset L(\mathcal{A})$. Moreover, if L' is ω-regular then so is L''.*

In the following, we construct from a given Büchi automaton $\mathcal{A}$ and an integer k, an automaton $\mathcal{A}_k$ that accepts those sequences in $L(\mathcal{A})$ that have an accepting $\mathcal{A}$-run in which an accepting state appears in every k-length block of consecutive states, thus $L(\mathcal{A}_k) \subseteq L(\mathcal{A})$ is a safety property.

Assume a Büchi automaton $\mathcal{A}: = (Q_A, \Sigma, \delta_A, q_A^0, F_A)$. Let $k > 0$ be an integer. We first define an ω-language $L_k(\mathcal{A})$, that is a subset of $L(\mathcal{A})$, where every string has an accepting run where accepting (F_A) states appear at least every k states from the beginning. Formally,

$$\begin{aligned} L_k(\mathcal{A}) \;=\; \{\sigma \in L(\mathcal{A}) : & \text{ for some accepting } \mathcal{A}\text{-run } r : q0, \ldots \text{ over } \sigma, \\ & \text{for every } i \geq 0, r[i \times k, ((i+1) \times k) - 1] \cap F_A \neq \emptyset\} \end{aligned}$$

Note that

$$\bigcup_{k>0} L_k(\mathcal{A}) \subseteq L(\mathcal{A})$$

This containment may, in general, be strict.

In general, $L_k(\mathcal{A})$ may not be contained in $L_{k+1}(\mathcal{A})$. However, it is not difficult to show that, if $k' \geq 2k - 1$ then $L_k(\mathcal{A}) \subseteq L_{k'}(\mathcal{A})$. As a consequence, by increasing k, we can get larger and larger safety properties contained in $L(\mathcal{A})$.

We next describe the construction of a DBA $\mathcal{A}_k$ that accepts the language $L_k(\mathcal{A})$. The construction of the automaton is an extension of the standard subset construction combined with partitioning the input into segments of length k. The segment partitioning is done by means of a modulo k counter. The automaton $\mathcal{A}_k$ simulates the possible runs of $\mathcal{A}$ on the input, and maintains the set of states that $\mathcal{A}$ may be at after reading each prefix. With each such state, $\mathcal{A}_k$ also keeps a bit, called *accepting state* bit, which indicates if an accepting state had been reached since the beginning of the most recent input segment.

Let $R = F_A \times \{1\} \cup (Q_A \setminus F_A) \times \{0, 1\}$. Fix some $k > 0$. Define the Büchi automaton $\mathcal{A}_k = (Q', \Sigma, \delta', q_0', F')$ where:

- $Q' = 2^R \times \{0, 1, ..., k-1\}$;
- $q_0' = \begin{cases} (\{(q_A^0, 0)\}, k-1) \text{ if } q_A^0 \notin F_A \\ (\{(q_A^0, 1)\}, k-1) \text{ otherwise} \end{cases}$
- F' is the set of all Q''s states whose first coordinate is non-empty, i.e., $F' = (2^R - \emptyset) \times \{0, 1, ..., k-1\}$

To define δ', we use two auxiliary transition functions $\beta, \gamma : 2^R \times \Sigma \rightarrow 2^R$ defined below. The function β captures the behavior of $\mathcal{A}_k$ within a segment: Note that the first coordinate of a state $q' \in Q'$ is a set of the form $\{(q_i, b_i) \ : 1 \leq i \leq m\}$ where each q_i is a state $\mathcal{A}$ can be in after reading a prefix, and b_i is the accepting bit which is 1 iff an accepting state was reached since the beginning of that segment. After reading an input letter s from a state q', $\mathcal{A}_k$ reaches all the states $\mathcal{A}$ reaches from q_i after reading s (i.e., $\delta(q_i, s)$), and the accepting bit is 1 if either it was 1 before (i.e., $b_i = 1$), or the state that is reached is accepting. Let $\beta\colon 2^R \times \Sigma \rightarrow 2^R$ be defined by:

$$\beta(\cup_{i=1}^{m}\{(q_i, b_i)\}, s) = \{(q, b) : \exists i. 1 \leq i \leq m \wedge q \in \delta(q_i, s) \wedge b \leftrightarrow (q \in F_A \vee b_i = 1)\}$$

The second auxiliary transition system, $\gamma\colon 2^R \times \Sigma \rightarrow 2^R$, captures the behavior of $\mathcal{A}$ when moving in between segments. It is similar to β, only that it restricts moves between segments to be only from states whose b_i is 1.

$$\gamma(\cup_{i=1}^{m}\{(q_i, b_i)\}, s) = \{(q, b) : \exists i. 1 \leq i \leq m \wedge b_i = 1 \wedge q \in \delta(q_i, s) \wedge (b \leftrightarrow q \in F_A)\}$$

We now define δ'. For $c > 0$,

$$\delta'\big(\langle \cup_{i=1}^{m}\{(q_i, b_i)\}, c\rangle,\ s\big) \ = \ \{\big(\beta(\cup_{i=1}^{m}\{(q_i, b_i)\}, s),\ c - 1\big)\}$$

and for $c = 0$,

$$\delta'\big(\langle \cup_{i=1}^{m}\{(q_i, b_i)\}, 0\rangle,\ s\big) \ = \ \{\big(\gamma(\cup_{i=1}^{m}\{(q_i, b_i)\}, s),\ k - 1\big)\}$$

Lemma 2. *$L_k(\mathcal{A})$ is a safety property and $L(\mathcal{A}_k) \ = L_k(\mathcal{A})$.*

Proof. Note that in the automaton $\mathcal{A}_k$, there are no transitions from states in $Q' - F'$ to states in F'. Thus the states of the A_k are partitioned into *good* states (i.e., members of F') and *bad* states (i.e., members of $Q' - F'$), so that an input sequence is accepted by it iff the unique run of A_k on the input contains only good states. From [11] it follows that $L(A_k)$ is a safety property. It remains to show that $L(\mathcal{A}_k) \ = L_k(\mathcal{A})$.

$\supseteq$: Assume $\sigma \in L_k(\mathcal{A})$. Thus, there exists an accepting $\mathcal{A}$-run $r = q^0, \ldots$ such that for each $j \geq 0$, $r[jk, (j+1)k] \cap F_A \neq \emptyset$. Let $\hat{r} : (R_0, c_0), (R_1, c_1), \ldots$ be the $\mathcal{A}_k$-run on σ. For every i, let $Q_i \subseteq Q_A$ be the set $\{q \ : (q, b) \in R_i$ for $b = 0$ or $b = 1\}$;i.e., Q_i is the set consisting of projections of each pair in R_i on its first component. By a simple induction, it can be shown that for $i \geq 0$, $q^i \in Q_i$. Since each Q_i is non-empty, $\hat{r}$ is an accepting run of $\mathcal{A}_k$. It therefore follows that $\sigma \in L(\mathcal{A}_k)$.

$\subseteq$: Assume $\sigma = s_1, \ldots$ is in $L(\mathcal{A}_k)$. Let $\hat{r} : (R_0, c_0), (R_1, c_1), \ldots$ be $\mathcal{A}_k$'s run on σ. For every i, let $Q_i \subseteq Q_A$ be the set $\{q \ : (q, b) \in R_i$ for $b = 0$ or $b = 1\}$. Since $\hat{r}$ is accepting, $Q_i \neq \emptyset$ for every $i \geq 0$. Define an infinite tree whose nodes are elements of the form $(q, j) \in Q_A \times \mathbb{N}$. The root of the tree is $(q_A^0, 0)$. For a tree node $n = (q, j)$, the children of n are the nodes $(n', j+1)$ such that $n' \in \delta_A(q, s_{j+1}) \cap Q_{j+1}$. Note that since $\hat{r}$ is an accepting $\mathcal{A}_k$-run, this tree is an infinite tree. Since it is finitely branching, form Köning's lemma, it follows that the tree has an infinite path r which is an $\mathcal{A}$-run. Moreover, from the way the accepting bits b_j are updated, it follows that, for every $i \geq 0$, the finite sequence $r[ik, (i+1)k]$ contains at least one occurrence of an F_A-state. Consequently, r is an accepting$\mathcal{A}$- run. It therefore follows that $\sigma \in L_k(\mathcal{A})$. □

Note. There are alternate ways of deriving safety properties contained in $L(\mathcal{A})$. We chose the one above for its relative simplicity.
We can now prove Lemma 1:

Proof (of Lemma 1). Assume $L' \subset L(\mathcal{A})$ is a safety property. Since $L(\mathcal{A})$ is not a safety property, there exists some $\sigma \in \Sigma^\omega \setminus L(\mathcal{A})$ such that $Pref(\sigma) \subseteq Pref(L(\mathcal{A})))$. Since $L' \subset L(\mathcal{A})$, it is the case, σ is not in L'. Since L' is safety property, there exists some $\alpha \in Pref(\sigma)$ which is a bad prefix for L', i.e., $\alpha\Sigma^\omega \cap L' = \emptyset$. Consider now the set $L_\alpha = \alpha\Sigma^\omega \cap L(\mathcal{A})$ of the $L(\mathcal{A})$ ω-strings with prefix α. The set L_α is an infinite ω-regular set. Let $\mathcal{B}$ be the Büchi automaton that accepts it, i.e., $L(\mathcal{B}) = L_\alpha$. Let k be an integer greater than or equal to the number of states in $\mathcal{B}$, and consider the language $L_k(\mathcal{B})$. From Lemma 2 it follows that $L_k(\mathcal{B})$ is a ω-regular safety property contained in L_α. Since k is at least as large as the number of states of $\mathcal{B}$, $L_k(\mathcal{B}) \neq \emptyset$. Since L' and $L_k(\mathcal{B})$ are safety properties, from [11], we see that $L'' = L' \cup L_k(\mathcal{B})$ is also a safety property. From the fact that L' and $L_k(\mathcal{B})$ are disjoint subsets of $L(\mathcal{A})$ and $L_k(\mathcal{B}) \neq \emptyset$, it follows that $L' \subset L'' \subset L(\mathcal{A})$. Note that since ω-languages are closed under union, it follows that if L' is ω-regular then so is L''. □

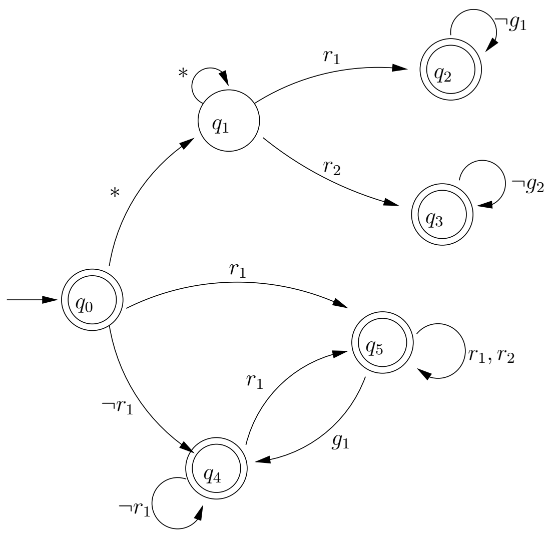

Fig. 1. Automaton for $\neg\Phi_I \vee \Phi$

In Fig 1 we give the automaton $\mathcal{A}$ for the property $\neg\Phi_I \vee \Phi$ for the permission manager example,i.e., example 1. Notice that this is a non-deterministic Buchi automaton. All double circles indicate accepting states and the state with an incoming edge from outside is the initial state. The input alphabet is $\{r_1, g_1, r_2, g_2\}$. The $*$ symbol on an input transition indicates that this transition can take place on any input symbol, i.e., it represents four transitions corresponding to each of the input symbols. The $\neg g_1$ symbol

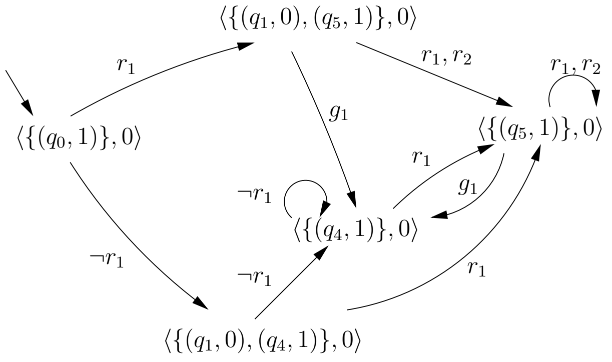
$\langle\{(q_1,0),(q_5,1)\},0\rangle$
r_1
r_1, r_2
r_1, r_2
g_1
$\langle\{(q_0,1)\},0\rangle$
$\langle\{(q_5,1)\},0\rangle$
r_1
g_1
$\neg r_1$
$\langle\{(q_4,1)\},0\rangle$
$\neg r_1$
$\neg r_1$
r_1
$\langle\{(q_1,0),(q_4,1)\},0\rangle$

Fig. 2. Our safety property: the $k = 1$ approximation

on a transition indicates that this transition can take place on any input symbol other than g_1, thus it represents three transitions. The symbols $\neg g_2, \neg r_1$ are similarly used.

In Fig. 2 we see the $k = 1$ approximation for our permission manager example, i.e., the automaton $\mathcal{A}_1$. This is a deterministic automaton. Each state of this automaton has the structure as given in the definition. There is an additional state, not shown in the figure, which is $\langle\emptyset, 0\rangle$. All unspecified transitions in the figure go to this state. For example, there is a transition from the state $\langle\{(q_5, 1)\}, 0\rangle$ to the state on input g_2. All states excepting $\langle\emptyset, 0\rangle$ are accepting states. It is not difficult to see that $L(\mathcal{A}) = L(\mathcal{A}_1)$. Note that this does not contradict Lemma 1 since $L(\mathcal{A})$ is a safety property.

4 Synthesis of Safety Properties by Bounded Automata

Section 3 describes the construction of deterministic finite state automata that synthesize safety properties contained in $L(\mathcal{A})$. It is often the case that "interesting" safety properties that are contained in $L(\mathcal{A})$ cannot be captured by finite state automata. For example, suppose that $L(\mathcal{A})$ is the set of ω-strings over $\{a, b\}$ where a appears infinitely often. Using the construction of Section 3, each $L(\mathcal{A}_k)$ requires that the number of bs between successive occurrences of a be bounded by a constant. Thus, they all rule out, e.g., a sequence where the number of b's between the i^{th} and the $(i+1)^{st}$ occurrence of a is i. On the other hand, an infinite state automaton that dynamically changes the bound on the number of input symbols before an acceptance state of $\mathcal{A}$ occurs on a run, can accept such a sequence.

Let $\mathcal{A} = (Q_A, \Sigma, \delta_A, q_A^0, F_A)$ be a Büchi automaton which we fix for this section. We generalize the construction of Subsection 3.1 using a class of infinite-state automata called *bounded automata*, and show that the language accepted by each bounded automaton is a safety property that is contained in $L(\mathcal{A})$. We also prove the converse, showing that every safety property contained in $L(\mathcal{A})$ is accepted by some bounded automaton.

Assume some (possibly infinite) set Y_B. A *bounded automaton* $\mathcal{B}$ is described by a tuple $(Q_B, \Sigma, \delta_B, q_B^0, F_B)$ where:

- $Q_B \subseteq Y_B \times 2^{Q_A \times \{0,1\}} \times (\mathbb{N} \cup \{\infty\})$ is a set of *states*;
- Σ is a finite *alphabet*;
- $\delta_B \colon Q_B \times \Sigma \to Q_B$ is a *transition function*. We further require that for every $\langle r, C, i\rangle, \langle r', C', i'\rangle \in Q_B$ and $a \in \Sigma$, if $\delta_B(\langle r, C, i\rangle, a) = \langle r', C', i'\rangle$ then the following all hold:
  - If $i = \infty$ then $i' = \infty$.
  - If $i' < i$ then $C' = \{(q', b') : \exists.(q, b) \in C$ such that $q' \in \delta_A(q, a) \wedge (b' = 1) \Leftrightarrow (b = 1 \vee q' \in F_A)\}$.
  - If $i' \geq i$ then $C' = \{(q', b') : \exists.(q, 1) \in C$ such that $q' \in \delta_A(q, a) \wedge (b = 1 \Leftrightarrow q' \in F_A)\}$.
- q_B^0 is the initial state, and it is required to be of the form $(r, \{(q_A^0, b)\}, i)$ where $b' = 1 \Leftrightarrow q_A^0 \in F_A$ and $i \neq \infty$.
- $F_B = \{\langle r, C, i\rangle : C \neq \emptyset \wedge i \neq \infty\}$.

It is to be noted that the range of δ_B is Q_B (and not 2^{Q_B}). This is done to make the notation simple and it also makes a bounded automaton as a deterministic automaton. A run of $\mathcal{B}$, an accepting run, and the language accepted by $\mathcal{B}$ are defined just in the case of Büchi automata.

The definition δ_B implies that once the second component of a state in a run is empty, it remains so. Similarly, if the third component of a state in a run is ∞, then it remains so. Thus, once a run enters a state in $Q_B \setminus F_B$, it remains there. (It thus follows that it suffices to define a run as accepting if it never reaches a $Q_B \setminus F_B$-state.) From [11] it follows that:

Lemma 3. *For a bounded automata $\mathcal{B}$, $L(\mathcal{B})$ is a safety property.*

Intuitively, given an input string, $\mathcal{B}$ simulates $\mathcal{A}$. Suppose $\mathcal{B}$ reaches a state $\langle r, C, i\rangle$. For each $(q, b) \in C$, q is a state $\mathcal{A}$ reaches on some run on the input seen thus far. The integer i is an upper bound on the number of steps before an accepting state of $\mathcal{A}$ is reached on some run. There are two types of transitions in δ_B— *decreasing* and *non-decreasing* transitions— denoting, respectively, those transitions that decrease i and those that do not. In case of decreasing transitions, C is updated to be the successor states of the corresponding runs of $\mathcal{A}$. In the case of non-decreasing transitions, only those runs containing an accepting state of $\mathcal{A}$, since the last occurrence of a non-decreasing transition, are considered for updating C. The bit b for each $(q, b) \in C$ records whether an accepting state has been reached since the last occurrence of a non-decreasing transition.

Let α be a finite string over Σ. If $\delta_B^*(q_B^0, \alpha) = \langle r, C, i\rangle$, then for every infinite string $\sigma \in L(\mathcal{B})$ such that $\alpha \in \mathit{Pref}(\sigma)$, for some $j \leq i$, $\delta_A^*(q_A^0, \sigma_{|\alpha|+j}) \cap F_A \neq \emptyset$ (where σ_j denotes the prefix of σ of length j). Thus, i is an upper bound on the number of inputs before an accepting state is going to appear after α is read on some run of $\mathcal{A}$. From the description of the operation of $\mathcal{B}$, given above, it is not difficult to show the following lemma.

Lemma 4. *For any bounded automaton $\mathcal{B}$, $L(\mathcal{B}) \subseteq L(\mathcal{A})$.*

Proof. Let $\sigma = s_1, ...$ be a string in $L(\mathcal{B})$. Let $u = \langle r_0, C_0, i_0 \rangle, ..., \langle r_j, C_j, i_j \rangle, ...$ be the (unique) run of $\mathcal{B}$ on σ. For each $j \geq 0$, let $C_j = (Q_j, b_j)$. As in the proof of Lemma 1, define an infinite tree whose nodes are elements of the form $(q, j) \in Q_A \times \mathbb{N}$. The root of the tree is $(q_A^0, 0)$. For a tree node $n = (q, j)$, where $j \geq 0$, the children of n are the nodes $(n', j+1)$ such that $n' \in \delta_A(q, s_{j+1}) \cap Q_{j+1}$. This tree is infinite and hence has an infinite path. Every such path defines a run of $\mathcal{A}$ on σ. Let k_j be the number of non-decreasing transitions of $\mathcal{B}$ that occur in the finite run $u[0, j]$. From our earlier discussion, it should be easy to see that every path of length j from the root node of the above tree contains at least k_j nodes of the form (q', l) where $q' \in F_A$. Since the run u has infinite number of non-decreasing transitions appearing in it, it is the case that every infinite path in the tree contains infinite nodes of the form (q', l) where $q' \in F_A$. Hence each such path gives an accepting run of $\mathcal{A}$ on σ. Since there exists at least one such path, we see that $\sigma \in L(\mathcal{A})$. $\square$

We next show that for every safety property in $L(\mathcal{A})$ there exists a bounded automaton that accepts it.

Recall that for a Büchi automaton $\mathcal{A}$, $L_f(\mathcal{A})$ is the regular language defined by the regular version of $\mathcal{A}$. Let $S \subseteq L(\mathcal{A})$ be a safety property. Note that every sequence in S has infinite number of prefixes that are in $L_f(\mathcal{A})$. For a sequence $\sigma \in S$ and $\alpha \in \mathit{Pref}(\sigma)$, let $\mathit{min\_idx}(\sigma, \alpha) = \min\{|\beta| : \alpha;\beta \in L_f(\mathcal{A}) \cap \mathit{Pref}(\sigma)\}$. Note that if $\alpha \in L_f(\mathcal{A}) \cap \mathit{Pref}(\sigma)$, then $\mathit{min\_idx}(\sigma, \alpha) = 0$.

For any finite string $\alpha \in \Sigma^*$, let $Z(\alpha, S) = \{\mathit{min\_idx}(\sigma, \alpha) : \sigma \in S \text{ and } \alpha \in \mathit{Pref}(\sigma)\}$. Obviously, if $\alpha \in L_f(\mathcal{A}) \cap \mathit{Pref}(S)$, then $Z(\alpha, S) = \{0\}$. Also, if $\alpha \notin \mathit{Pref}(S)$ then $Z(\alpha, S) = \emptyset$. The following lemma establishes that $Z(\alpha, S)$ is always finite.

Lemma 5. *For any $\alpha \in \Sigma^*$, $Z(\alpha, S)$ is finite.*

Proof. From the comments above it suffices to prove the claim for the case when $\alpha \in \mathit{Pref}(S) \setminus L_f(\mathcal{A})$. Assume, by way of contradiction, that $Z(\alpha, S)$ is infinite. Since Σ is a finite set, it follows that there exists some $a_0 \in \Sigma$ such that $Z(\alpha a_0, S)$ is an infinite, hence $\alpha_1 = \alpha; a_0 \notin L_f(\mathcal{A})$. We can repeat this observation inductively, and obtain an infinite sequence of finite sequences $\alpha = \alpha_0, a_1, \ldots$ such that for every $i \geq 0$, $\alpha_{i+1} = \alpha_i; a_i$ for some $a_i \in \Sigma$, $Z(\alpha_i, S)$ is infinite, and $\alpha_i \notin L_f(\mathcal{A})$. Let $\beta \in \Sigma^\omega$ be the limit sequence of the α_i's. Since for every i, $\alpha_i \in \mathit{Pref}(S)$, and S is a safety property, it follows that $\beta \in S$. All the prefixes of β of length greater than $|\alpha|$ are not in $L_f(\mathcal{A})$. Consequently, $\beta \notin L(\mathcal{A})$. It therefore follows that $S \not\subseteq L(\mathcal{A})$, which is a contradiction. $\square$

For $\alpha \in \Sigma^*$, let $\mathit{idx}(\alpha, S) = \max_{i \in Z(\alpha, S)} i$. If $Z(\alpha, S)$ is empty, we define $\mathit{idx}(\alpha, S) = \infty$. Thus, $\mathit{idx}(\alpha, S) \in \mathbb{N}$ iff $\alpha \in \mathit{Pref}(S)$. With a safety property $S \subseteq L(\mathcal{A})$, we associate a bounded automaton $\mathcal{D} = (Q_D, \Sigma, \delta_D, q_D^0, F_D)$ where:

- Q_D consists of triples of the form $\langle \alpha, C, i \rangle$ where $\alpha \in \Sigma^*$, $C \subseteq Q_A \times \{0, 1\}$, and $i = \mathit{idx}(\alpha, S)$;
- For every $\langle \alpha, C, i \rangle \in Q_D$ and $a \in \Sigma$, $\delta_D(\langle \alpha, C, i \rangle, a) = \langle \alpha', C', i' \rangle$ implies $\alpha' = \alpha; a$.
- q_D^0 is the triple $\langle \epsilon, (q_A^0, b), \mathit{idx}(\epsilon, S) \rangle$ where ϵ is the empty sequence and $b = 1 \Leftrightarrow q_A^0 \in F_A$;

For $\alpha \in \Sigma^*$, it is easy to see that $\delta_D^*(q_D^0, \alpha)$ is of the form $(\alpha, C, idx(\alpha, S))$. Moreover, if $\alpha \in \mathit{Pref}(S)$, then $C \neq \emptyset$ and $i \neq \infty$. Thus, after having read any prefix of a S-sequence, $\mathcal{D}$ is in a F_D-state. Thus, $S \subseteq L(\mathcal{D})$. Conversely, if $\sigma \notin S$, then there exists some $\alpha \in \mathit{Pref}(\sigma) \setminus \mathit{Pref}(S)$. In this case, the state reached by $\mathcal{D}$ after reading α is of the form (α, C, ∞), and thus $\sigma \notin L(\mathcal{D})$. We thus have:

Lemma 6. $L(\mathcal{D}) = S$.

The following theorem follows from Lemma 4 and Lemma 6:

Theorem 1 (Completeness). *Let $\mathcal{A}$ be a Büchi automaton. Then every safety property $S \subseteq L(\mathcal{A})$ is accepted by some bounded automaton.*

Recursive and History Based Automata. We have shown that the class of languages accepted by bounded automata is exactly the class of safety properties contained in $L(\mathcal{A})$. We say that a bounded automaton $\mathcal{B} = (Q_B, \Sigma, \delta_B, q_B^0, F_B)$ is *recursive* if the set Q_B is recursive and δ_B is a computable function. It is to be noted that only recursive bounded automata are useful. It is not difficult to see that the automata $\mathcal{A}_k$ that we defined in Subsection 3.1 are recursive bounded automata as each of these is a finite state automaton. Recall that, in these automata, k is the length of the segments into which the input string is divided. We can generalize the automata A_k, so that it starts with an initial value of k and increases the value of k dynamically; that is, it increases the lengths of the segments according to some computable function f, so that $f(i)$ is the length of the i^{th} segment.

We can now define a class of recursive bounded automata, called *history based automata*: Let $f\colon \Sigma^* \times 2^{Q_A \times \{0,1\}} \rightarrow (\mathbb{N} \cup \{\infty\})$ be some computable function. A *history based automaton with respect to* f is the bounded automaton $\mathcal{B}_f = (Q_B, \Sigma, \delta_B, q_B^0, F_B)$ where $Q_B = \{(\alpha, C, f(\alpha, C)) : \alpha \in \Sigma^*,\ C \subseteq Q_A \times \{0,1\}\}$. Note that $\mathcal{B}_f$ is uniquely defined. Essentially, the bound in each state of $\mathcal{B}_f$ is defined by the recursive function f. It is not difficult to show that every recursive bounded automaton is homomorphic to a history based automaton.

5 Related Work

Some of the techniques we employ are somewhat reminiscent of techniques used for verifying that a safety property described by a state machine satisfies a correctness specification given by an automaton or temporal logic. For example, simulation relations/ state-functions together with well-founded mappings [5,1,12] have been proposed in the literature for this purpose. Our bounded automata use a form of well-founded mappings in the form of positive integer values that are components of each state. (This is as it should be, since we need to use some counters to ensure that an accepting state eventually appears.) However, here we are not trying to establish the correctness of a given safety property defined by a state machine, but rather, we are deriving safety properties that are contained in the language of an automaton.

In [7,8] Larsen et.al. propose a method for turning an implicit specification of a component into an explicit one. I.e., given a context specification (in their case a process algebraic expression with a hole, where the desired components needs to be plugged in) and an overall specification, they fully automatically derive a temporal safety property characterizing the set of all implementations which, together with the given context, satisfy the overall specification. While this technique has been developed for component synthesis, it can also be used for synthesizing optimal monitors in a setting where the interface specification Φ_I and the goal specification Φ are both safety properties. In this paper, we do not make any assumptions on Φ_I and Φ. They can be arbitrary properties specified in temporal logic or by automata. We are aiming at exploiting liveness guarantees of external components (contexts), in order to establish liveness properties of the overall system under certain additional safety assumptions, which we can run time check (monitor). This allows us to guarantee that the overall system is as live as the context, as long as the constructed monitor does not cause an alarm.

Perhaps closest to our work in motivation is the work in [10]. The approach taken there, however, is that of considering the interaction between the module and the interface as a 2-player game, where the interface has a winning strategy if it can guarantee that no matter what the module does, Φ is met while maintaining Φ_I. Run-time monitoring is used to verify that the interface has a winning strategy.

There has been much work done in the literature on monitoring violations of safety properties in distributed systems. In these works, the safety property is typically explicitly specified by the user. Our work is more on deriving safety properties from component specifications than developing algorithms for monitoring given safety properties. In this sense, the approach to use safety properties for monitoring that have been automatically derived by observation using techniques adapted from automata learning (see [4]) is closer in spirit to the proposal here. Much attention has since been spent in optimizing the automatic learning of the monitors [9]. However, the learned monitors play a different role: whereas the learned monitors are good, but by no means complete, sensors for detecting unexpected anomalies, the monitors derived with the techniques of this paper *imply* the specifying property as long as the guarantees of the component provider are true.

6 Conclusions and Discussion

In this paper, we considered the problem of customizing a given, off-the-shelf, reactive component to user requirements. In this process, we assume that the reactive module's external behavior is specified by a formula Φ_I and the desired goal specifications is given by a formula Φ. Both Φ_I and Φ can be arbitrary properties, i.e. , they need not be safety properties. We presented methods for obtaining a safety specification ϕ so that $\Phi_I \wedge \phi \rightarrow \Phi$. Our methods obtain ϕ as a deterministic (possibly infinite state) automaton. This automaton can be used to monitor execution of the off-the-shelf component so that it does not violate ϕ and hence satisfies the goal specification Φ.

There are a number of issues that need to be further addressed. When the desired property is given by a finite state automaton, then monitoring executions can be done in real time, i.e., each successive state change of the automaton can be done within a constant time that only depends on the size of the automaton but not on the length of the computation, i.e., the history seen thus far. On the other hand, when ϕ is given by an infinite state automaton, real time change in the state of the automaton may not always be achievable. For example, we defined a class of infinite state automata, called history based automata, that divide the input into segments and ensure that an appropriate liveness condition is satisfied in each segment. In these automata, the lengths of successive segments can vary dynamically and are computed as functions of the history using a computable function. In such cases, one has to ensure that the computation of the length of the next segment does not take too long a time. Of course, one can compute lengths of successive segments by simple functions such as increasing the lengths by a constant factor, etc. These and other issues need to be further investigated. We also need to further investigate practical cases where these techniques can be applied.

References

1. M. Abadi and L. Lamport. The existence of state mappings. In *Proceedings of the ACM Symposium on Logic in Computer Science*, 1988.
2. B. Alpern and F. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
3. E. A. Emerson and A. P. Sistla. Triple exponential decision procedure for the logic ctl*. In *Workshop on the Logics of Program, Carnegie-Mellon University*, 1983.
4. H. Hungar and B. Steffen. Behavior-based model construction. *STTT*, 6(1):4–14, 2004.
5. B. Jonsson. Compositional verification of distributed systems. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, 1987.
6. L. Lamport. Logical foundation, distributed systems- methods and tools for specification. *Springer-Verlag Lecture Notes in Computer Science*, 190, 1985.
7. K. Larsen. Ideal specification formalisms = expressivity + compositionality + decidability + testability + ... In *Invited Lecture at CONCUR 1990, LNCS 458*, 1990.
8. K. Larsen. The expressive power of implicit specifications. In *ICALP 1991, LNCS 510*, 1991.
9. T. Margaria, H. Raffelt, and B. Steffen. Knowledge-based relevance filtering for efficient system-level test-based model generation (to appear). *Innovations in Systems and Software Engineering, a NASA Journal, Springer Verlag.*
10. A. Pnueli, A. Zaks, and L. D. Zuck. Monitoring interfaces for faults. In *Proceedings of the 5^{th} Workshop on Runtime Verification (RV'05)*, 2005. To appear in a special issue of ENTCS.
11. A. P. Sistla. On characterization of safety and liveness properties in temporal logic. In *Proceedings of the ACM Symposium on Principle of Distributed Computing*, 1985.
12. A. P. Sistla. Proving correctness with respect to nondeterministic safety specifications. *Information Processing Letters*, 39:45–49, 1991.
13. M. Vardi, P. Wolper, and A. P. Sistla. Reasoning about infinite computations. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, 1983.