JAIST Repository
https://dspace.jaist.ac.jp/

Title	Application of DES Theory to Verification of Software Components
Author(s)	HIRAISHI, Kunihiko; KU ERA, Petr
Citation	IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences, E92-A(2): 604-610
Issue Date	2009-02-01
Type	Journal Article
Text version	publisher
URL	http://hdl.handle.net/10119/8517
Rights	Copyright (C)2009 IEICE. Kunihiko HIRAISHI, Petr KU ERA, IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences, E92-A(2), 2009, 604-610. http://www.ieice.org/jpn/trans_online/
Description	

JAIST
JAPAN ADVANCED INSTITUTE OF SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

PAPER

Application of DES Theory to Verification of Software Components

Kunihiko HIRAISHI[†a)], ***Member and*** **Petr KUČERA**[††], ***Nonmember***

SUMMARY Software model checking is typically applied to components of large systems. The assumption generation is the problem of finding the least restrictive environment in which the components satisfy a given safety property. There is an algorithm to compute the environment for properties given as a regular language. In this paper, we propose a general scheme for computing the assumption even for non-regular properties, and show the uniqueness of the least restrictive assumption for any class of languages. In general, dealing with non-regular languages may fall into undecidability of problems. We also show a method to compute assumptions based on visibly pushdown automata and their finite-state abstractions.
key words: *discrete event systems, software component verification, supervisory control*

1. Introduction

Model checking is widely used for determining whether a discrete-state system satisfies certain properties by exhaustively exploring all possible state transitions [1]. Software model checking is typically applied to components of large systems by several reasons, e.g., one can ignore the detailed behavior of the operating system in which a component runs; the size of the state space for the entire system becomes too large, etc. [2].

Let M be a software component and let E be the environment of M (i.e., software components other than M). Then the entire software system is denoted by the composition of M and E, written by $M||E$. Let P be a safety property defined on $M||E$. Since the component M is developed independently of other software components, M is usually built to satisfy P for all possible environment. However, this approach is overly pessimistic. *Assumption generation* is the problem of finding the least restrictive environment E^* so that P holds on $M||E^*$ [2]. If there exists an environment E in which the component M satisfies the property, then such E^* exists and all the behavior of the software allowed by E are also allowed by E^*. An algorithm to compute the least restrictive environment E^* was proposed for the case that both M and P are represented by finite automata [2]. The obtained assumption will be used for selecting and/or designing appropriate environment for the component. Moreover, assumption generation may be seen as a way of providing automated support for assume-guarantee reasoning for software verification [3].

Since the behavior of a software is not necessarily regular and is usually modeled by a context-free language, assumption generation for finite-state systems is not sufficient. In this paper, we study the assumption generation problem for the case that the property P is represented by a non-regular language.

We first propose a general scheme for computing the assumption even for non-regular properties. The scheme is based on the supervisory control theory of discrete event systems. By simple applications of the results in the supervisory control theory, we can show that the least restrictive environment uniquely exists for any classes of languages, and is computed by application of several language operations. In general, dealing with non-regular languages may fall into undecidability of problems. Next we propose a method to compute assumptions based on finite-state abstraction of automata. Since complements of languages are abstracted in the method, the obtained assumption may still be non-regular. Finally, as a dual problem of finding least restrictive assumption for safety property, we propose a scheme for generating assumptions from liveness properties.

2. Assumption Generation Problem

2.1 Modeling Software Components

One of main concerns in component-based software development is how to check correctness of component composition and component use. A component can be viewed as a black-box entity which provides and/or requires a set of services. To describe communication behavior between software components in a formal way, various approaches have been taken [4]–[8]. In many researches dealing with software component verification, state transition based modeling, such as finite-state machines and process algebra, is used for describing interface and protocols of software components. Such an abstracted view of software systems is possible because correctness of individual component is checked separately and we can focus only on interface part of software components. Once formal models for communication behavior of software components are built, formal verification techniques such as model checking and theorem proving can be applied.

Manuscript received August 8, 2008.
Manuscript revised October 30, 2008.
[†]The author is with the School of Information Science, Japan Advanced Institute of Science and Technology, Nomi-shi, 923-1292 Japan.
[††]The author is with Faculty of Mathematics and Physics, Charles University, Czech Republic.
a) E-mail: hira@jaist.ac.jp
DOI: 10.1587/transfun.E92.A.604

Copyright © 2009 The Institute of Electronics, Information and Communication Engineers

2.2 Automata-Based Modeling

In this paper, automata are used for describing behavior of software like in [2], [9]. *An automaton* is a 5 tuple $\mathcal{M} = (Q, \Sigma, \delta, q_0, Q_m)$, where Q is the set of states, Σ is the set of events, $\delta : Q \times \Sigma \to Q$ is the transition function, q_0 is the initial state, and $Q_m \subseteq Q$ is the set of marked states. The state transition function δ is defined as a partial function and we will write $\delta(q, \sigma)!$ to indicate that "$\delta(q, \sigma)$ is defined". As usual, the domain of δ is extended to $Q \times \Sigma^*$. The set of trajectories generated by $\mathcal{M}$ is $L(\mathcal{M}) := \{s \in \Sigma^* \mid \delta(q_0, s)!\}$, and the set of trajectories marked by $\mathcal{M}$ is $L_m(\mathcal{M}) := \{s \in \Sigma^* \mid \delta(q_0, s) \in Q_m\}$. Sequences in $L_m(\mathcal{M})$ are related to completion of tasks, but we do not use them in this paper. When Q is finite, we call $\mathcal{M}$ *a finite automaton*.

Let $\overline{s}$ denote the set of all prefixes of a sequence s, and let $\overline{L}$ denotes the prefix closure of a language L. Moreover, for a subset Σ' of Σ, let $\mathcal{P}_{\Sigma'}$ denote *the projection operator* defined for each $s \in \Sigma^*$, $\mathcal{P}_{\Sigma'}(s)$ is the sequence obtained by removing all symbols in $\Sigma - \Sigma'$ from s. For a language $L \subseteq \Sigma^*$, let $\mathcal{P}_{\Sigma'}(L) := \{\mathcal{P}_{\Sigma'}(s) \mid s \in L\}$. When we simply write $\mathcal{P}(\cdot)$, it means the projection operator $\mathcal{P}_{\Sigma_o}(\cdot)$ to the set of *observable events* Σ_o, which will be introduced later. *The inverse projection operator* is defined by $\mathcal{P}^{-1}_{\Sigma'}(L) := \{s \in \Sigma^* \mid \mathcal{P}_{\Sigma'}(s) \in L\}$ and $\mathcal{P}^{-1}(L) := \{s \in \Sigma^* \mid \mathcal{P}(s) \in L\}$.

Given two automata $\mathcal{M}_i = (Q_i, \Sigma_i, \delta_i, q_{0i}, Q_{m,i})$, $i = 1, 2$, *the composition* of $\mathcal{M}_1$ and $\mathcal{M}_2$, denoted by $\mathcal{M}_1 \| \mathcal{M}_2$, is defined as the automaton $(Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{01}, q_{02}), Q_{m1} \times Q_{m2})$, where for each $q = (q_1, q_2) \in Q$ and $\sigma \in \Sigma_1 \cup \Sigma_2$:

$$\delta(q,\sigma) := \begin{cases} (\delta_1(q_1,\sigma), \delta_2(q_2,\sigma)) & \text{if } \delta_i(q_i,\sigma)!\ (i=1,2) \wedge \\ & \quad \sigma \in \Sigma_1 \cap \Sigma_2; \\ (\delta_1(q_1,\sigma), q_2) & \text{if } \delta_1(q_1,\sigma)! \wedge \sigma \notin \Sigma_2; \\ (q_1, \delta_2(q_2,\sigma)) & \text{if } \delta_2(q_2,\sigma)! \wedge \sigma \notin \Sigma_1; \\ \text{undefined} & \text{otherwise.} \end{cases}$$

2.3 Assumption Generation for Safety Properties

Let M be an automaton over Σ_M representing a software component and let E be an automaton over Σ_E representing an environment. $C := \Sigma_M - \Sigma_E$ is called the set of *component events* of M, and $\mathcal{I} := \Sigma_M \cap \Sigma_E$ is called the set of *interface events*. We assume that the environment cannot control and observe all events in C, as assumed in [2]. The environment controls each component (i.e., the environment calls procedures of components and gets results) only by interface events.

A safety property is a kind of properties like "*something bad never happen*", and is represented by an automaton P over Σ_P. The behavior of the system is required to be within the range of P.

Definition 2.1: Given a software component M and a safety property P, *assumption generation for safety properties* (AGS) is the problem of finding an automaton A over $\Sigma_A = \mathcal{I} \cup (\Sigma_P - C)$ such that $M \| A$ satisfies P, i.e., $\mathcal{P}_{\Sigma_P}(L(M \| A)) \subseteq L(P)$ holds. Since $M \| P$ and $M \| A$ have the same set of events $\Sigma_M \cup \Sigma_P$, this condition is rewritten as $L(M \| A) \subseteq L(M \| P)$.

2.4 Supervisory Control of Discrete Event Systems

As usual, the set Σ of events is partitioned into two disjoint sets Σ_c and Σ_{uc}, where Σ_c is the set of *controllable events* and Σ_{uc} is the set of *uncontrollable events*. Similarly, Σ is partitioned into Σ_o and Σ_{uo}, where Σ_c is the set of *observable events* and Σ_{uc} is the set of *unobservable events*.

Definition 2.2: Given an automaton G over Σ and a prefix-closed language $K \subseteq L(G)$, the supervisory control and observation problem ($SCOP$) is the problem of finding a function $S : \mathcal{P}(L(G)) \to 2^{\Sigma_c}$ such that $L(S/G) = K$, where $L(S/G)$ is the smallest set such that: (i) $\varepsilon \in L(S/G)$ and (ii) $s \in L(S/G) \wedge s\sigma \in L(G) \wedge \sigma \notin S(\mathcal{P}(s)) \Rightarrow s\sigma \in L(S/G)$.

The function S is called *a supervisor*, and determines a set of events to be disabled. If there is no such supervisor, the next objective is to find a relaxation of K, i.e., a subset of K that satisfies the requirements. The relaxation is characterized by *the supremal controllable and observable sublanguage* of K [10], [11]. When $\Sigma_c = \Sigma_o$, the function S can be represented by an automaton, and the problem is simply rewritten as follows: Given an automaton G over Σ and a prefix-closed language $K \subseteq L(G)$, find an automaton S defined over $\Sigma_c = \Sigma_o$ such that $L(G \| S) \subseteq K$, where $L(G \| S)$ is the largest it can be.

When $\Sigma_c \subseteq \Sigma_o$, i.e., all controllable events are observable, observability and *normality* are equivalent provided that the language is controllable [10], [11]. Moreover, it was shown that *the supremal controllable and normal sublanguage* of a given language always exists. The definition of controllability and normality will be shown later.

2.5 Reformulation of AGS as $SCOP$

To relate Problem AGS with results on the supervisory control scheme, we define the completion of automata. Let $\mathcal{M} = (Q, \Sigma_{\mathcal{M}}, \delta, q_0, Q_m)$ be an automaton with $\Sigma_{\mathcal{M}} \subseteq \Sigma$. *The completion* of $\mathcal{M}$ w.r.t. $\Sigma \supseteq \Sigma_{\mathcal{M}}$ is the automaton $\mathcal{M}^{\uparrow\Sigma} = (Q, \Sigma, \delta^{\uparrow\Sigma}, q_0, Q_m)$ such that (i) for each $\sigma \in \Sigma_{\mathcal{M}}$, $\delta^{\uparrow\Sigma}(q, \sigma) = q'$ iff $\delta(q, \sigma) = q'$; (ii) for each $\sigma \in \Sigma - \Sigma_{\mathcal{M}}$ and $q \in Q$, $\delta^{\uparrow\Sigma}(q, \sigma) = q$. Intuitively, the completion of an automaton is obtained by adding self-loops with events in $\Sigma - \Sigma_{\mathcal{M}}$ to each state. $L(\mathcal{M}^{\uparrow\Sigma})$ contains all trajectory $s \in \Sigma^*$ such that $\mathcal{P}_{\Sigma_{\mathcal{M}}}(s) \in L(\mathcal{M})$.

Let $\Sigma := \Sigma_M \cup \Sigma_A = \Sigma_M \cup \Sigma_P$ be the set of all events. Given an instance of Problem AGS, we construct an instance of Problem $SCOP$ as follows: G is the completion of M w.r.t. Σ, $K = L(M \| P)$, and $\Sigma_{uc} = \Sigma_{uo} = C$. Then the least restrictive environment is obtained as the supervisor A such that $L(M^{\uparrow\Sigma} \| A)$ is the supremal controllable and normal sublanguage of K.

In the formulation of $SCOP$, the supervisor can manipulate only events of the plant G. Since the assuption A may

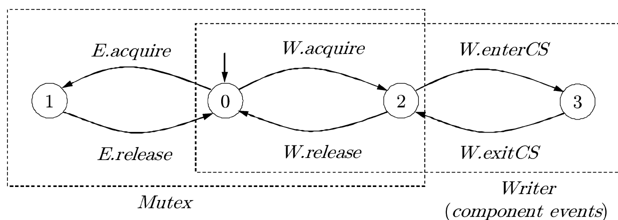

Fig. 1 Software component $M = Mutex||Writer$.

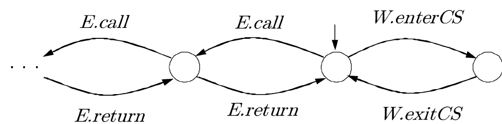

Fig. 2 A safety property.

contain events other than events of M, we need to use $M^{\uparrow\Sigma}$ as the plant.

Example 1: We show an instance of Problem *AGS*. Figure 1 shows a software component consisting of two subcomponents *Mutex* and *Writer*, where *Mutex* works for mutual exclusion between the component and the environment, and subcomponent *Writer* writes data on common resources. We specify the set of component events C and the set of interface event $\mathcal{I}$ as follows:

$$C = \{W.acquire, W.release, W.enterCS, W.exitCS\},$$
$$\mathcal{I} = \{E.acquire, E.release\}.$$

Note that this component is the same as that used in [2]. As a non-regular safety property P, we give an automaton shown in Fig. 2.

Since the supremal controllable and observable sublanguage of a given language always exists, we have the following theorem.

Theorem 2.3: If Problem *AGS* has a solution, then the problem has the largest solution w.r.t. language inclusion.

We remark that this result is independent of the class of languages.

2.6 Procedures to Compute the Supremal Controllable and Normal Sublanguage

Definition 2.4: Given an automaton G over Σ and a prefix-closed language $K \subseteq L(G)$,

- K is called *controllable* w.r.t. $L(G)$ and Σ_{uc} if $K\Sigma_{uc} \cap L(G) \subseteq K$,
- K is called *normal* w.r.t. $L(G)$ and Σ_o if $K = \mathcal{P}^{-1}(\mathcal{P}(K)) \cap L(G)$.

The supremal controllable and normal sublanguage of a prefix-closed language $K \subseteq L(G)$, denoted by $sup\,CN(K)$, is computed through the following procedures [10], [11]:

- $sup\ CN(K) = sup\ N(sup\ C(K))$ (see Remark 4.3 of [10]), where $supN(\cdot)$ denotes the supremal normal sublanguage and $sup\ C(\cdot)$ denotes the supremal controllable sublanguage. Note that if K is prefix-closed, than so are $sup\ N(K)$ and $sup\ C(K)$.
- Suppose that K is prefix-closed. $sup\,C(K)$ is computed by performing the following iterative procedure until $K_{i+1} = K_i$:
  - $K_0 := K$;
  - $K_{i+1} := K_i - ((L(G) - K_i)/\Sigma_{uc})\Sigma^*$,

  where $L_1/L_2 := \{s \in \Sigma^* \mid \exists t \in L_2.st \in L_1\}$ is *the quotient operation*.
- Suppose that K is prefix-closed. Then $sup\,N(K) = K - \mathcal{P}^{-1}(\mathcal{P}(L(G) - K))\Sigma^*$.

The language operations used in the above procedures are $L_1 - L_2$ (difference), L_1L_2 (concatenation), L_1/L_2 (quotient), $\mathcal{P}$ (projection), and $\mathcal{P}^{-1}$ (inverse projection). Therefore, we have the following theorem on the decidability of Problem *AGS*.

Theorem 2.5: If the class of languages representing $L(M)$ and $L(P)$ is closed under difference, concatenation, quotient, projection and inverse projection, then Problem *AGS* is decidable.

3. Representation of Non-regular Properties

We use a class of automata, called *visibly pushdown automata* [13], [14], as formalism for representing non-regular properties.

3.1 Visibly Pushdown Automata

A *pushdown alphabet* is a tuple $\widehat{\Sigma} = \langle\Sigma_{call}, \Sigma_{ret}, \Sigma_{int}\rangle$, where Σ_{call} is a set of *call* symbols, Σ_{ret} is a set of *return* symbols, and Σ_{int} is a set of *internal* symbols. By Σ, we denote the set of all symbols, i.e. $\Sigma = \Sigma_{call} \cup \Sigma_{ret} \cup \Sigma_{int}$.

A *visibly pushdown automaton* (*VPA*) over $\widehat{\Sigma}$ is a tuple $\mathcal{M} = (Q, \widehat{\Sigma}, \Gamma, \delta, q_0, Q_F)$, where Q is the nonempty finite set of states, $q_0 \in Q$ is the initial state, Γ is the finite stack alphabet that contains a special bottom-of-stack symbol $\perp$, $\delta = \delta_{call} \cup \delta_{ret} \cup \delta_{int}$ is the transition function, where $\delta_{call} : Q \times \Sigma_{call} \to Q \times (\Gamma - \{\perp\})$, $\delta_{ret} : Q \times \Sigma_{ret} \times \Gamma \to Q$, and $\delta_{int} : Q \times \Sigma_{int} \to Q$, and $Q_F \subseteq Q$ is the set of final states. A VPA $\mathcal{M}$ over $\widehat{\Sigma}$ is said to be a $\widehat{\Sigma}$-VPA.

Intuitively, a visibly pushdown automaton is a pushdown automaton, which is restricted in such a way that it pushes onto stack only when it reads a call symbol, it pops from stack when it reads a return symbol, and it does not use the stack when it reads an internal symbol. Let $St = (\Gamma - \{\perp\})^*\{\perp\}$. A *configuration* is a pair (q, μ), where $q \in Q$ and $\mu \in St$. As usual, we extend the domain and the range of transition function δ to $\delta : (Q\times St)\times\Sigma^* \to (Q\times St)$ (see the detail in [13], [14]).

A sequence $s \in \Sigma^*$ is *accepted* by a $\widehat{\Sigma}$-VPA $\mathcal{M}$, if there exists $\mu \in St$ such that $\delta((q_0, \perp), s) = (q, \mu) \wedge q \in Q_F$. The

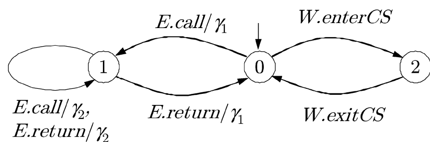

Fig. 3 VPA P representing the safety property.

language of $\mathcal{M}$, denoted by $L(\mathcal{M})$, is the set of sequences accepted by $\mathcal{M}$. A language over finite sequences $L \subseteq \Sigma^*$ is a *visibly pushdown language* (VPL) w.r.t. $\widehat{\Sigma}$ (a $\widehat{\Sigma}$-VPL) if there is a $\widehat{\Sigma}$-VPA $\mathcal{M}$ such that $L(\mathcal{M}) = L$.

In order to represent generated languages in VPAs, we can assume that there exists a state q_{ud} representing the destination of undefined transitions, and let $Q_F := Q - \{q_{ud}\}$. In diagrams of VPAs, we will not draw the state q_{ud} together with connected arrows.

Example 2: The safety property shown in Fig. 2 is represented by a VPA P in Fig. 3, defined over

$$\Sigma_{call} = \{E.call\},$$
$$\Sigma_{ret} = \{E.return\},$$
$$\Sigma_{int} = \{W.enterCS, W.exitCS\},$$

where

- each transition $q_i \overset{a}{\rightarrow} q_j$ means that $\delta_{int}(q_i, a) = q_j$ if $a \in \Sigma_{int}$ (on reading an internal symbol a, the control state changes from q to q');
- each transition $q_i \overset{a/\gamma}{\rightarrow} q_j$ means that $\delta_{call}(q_i, a) = (q_j, \gamma)$ if $a \in \Sigma_{call}$ (on reading a call symbol a, γ is pushed onto stack and the control state changes from q to q'); $\delta_{ret}(q_i, a, \gamma) = q_j$ if $a \in \Sigma_{ret}$ (on reading a return symbol a, γ is popped from the top of stack and the control state changes from q to q').

3.2 Closure Properties of VPLs

The following theorem shows, that the class of visibly pushdown languages has similar closure as the class of regular languages.

Theorem 3.1: The class of visibly pushdown languages is closed under union, intersection, renaming, concatenation and Kleene-* [13].

The theorem was proved in a constructive way, i.e., given a VPA, we can effectively compute the resulting VPA by each operator.

Let $\mathcal{M} = (Q, \widehat{\Sigma}, \Gamma, \delta, q_0, Q_F)$ be a visibly pushdown automaton. A $\widehat{\Sigma}$-VPA representing complementation of $L(\mathcal{M})$ can be produced from $\mathcal{M}$ by simply setting the set of final states to $Q - Q_F$. The difference $L_1 - L_2$ is computed by the intersection of L_1 and the complementation of L_2. Hence, the class of VPLs is also closed under difference.

Unfortunately, the class of VPLs is not closed under projection. Let us consider a pushdown alphabet $\widehat{\Sigma}$ with $\Sigma_{call} = \{a, b\}$, $\Sigma_{ret} = \{c\}$, and $\Sigma_{int} = \emptyset$ and a language $L = (ab)^n c^{2n}$, which is trivially a visibly pushdown language. If, however, b is the only unobservable event, then $\mathcal{P}(L) = a^n c^{2n}$, which is not a visibly pushdown language. Hence, we cannot use the result of Theorem 2.5 without any additional assumptions.

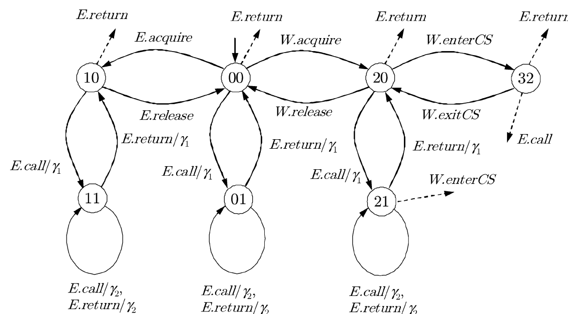

Fig. 4 VPA $M \| P$.

Similarly, the class of VPLs is not closed under inverse projection. With the same pushdown alphabet $\widehat{\Sigma}$ as above, a language $L' = a^n c^n$ is a visibly pushdown language, but $\mathcal{P}^{-1}(L') = b^*(ab^*)^n(cb^*)^n$ cannot be accepted by any $\widehat{\Sigma}$-VPA.

Given two automata $\mathcal{M}_i = (Q_i, \Sigma_i, \delta_i, q_{0i}, Q_{m,i})$, $i = 1, 2$, the language of composition of $\mathcal{M}_1$ and $\mathcal{M}_2$, $L(\mathcal{M}_1 \| \mathcal{M}_2)$, is defined as $\mathcal{P}^{-1}_{\Sigma_1}(L(\mathcal{M}_1)) \cap \mathcal{P}^{-1}_{\Sigma_2}(L(\mathcal{M}_2))$, where $\Sigma = \Sigma_1 \cup \Sigma_2$. This means that $L(\mathcal{M}_1 \| \mathcal{M}_2)$ is not necessarily a $\widehat{\Sigma}$-VPL even if $\mathcal{M}_1$ and $\mathcal{M}_2$ are $\widehat{\Sigma}$-VPAs.

We can show the following positive result for a restricted case.

Lemma 3.2: Suppose that all unobservable events correspond to internal symbols of a pushdown alphabet $\widehat{\Sigma}$.

1. If L is a $\widehat{\Sigma}$-VPL, then $\mathcal{P}(L)$ is also a $\widehat{\Sigma}$-VPL.
2. If L is a $\widehat{\Sigma}$-VPL consisting only of observable events, then $\mathcal{P}^{-1}(L)$ is also a $\widehat{\Sigma}$-VPL.

Proof. Given a $\widehat{\Sigma}$-VPA $\mathcal{M}$ with $L(\mathcal{M}) = L$, we obtain a $\widehat{\Sigma}$-VPA $\mathcal{M}'$ with $L(\mathcal{M}') = \mathcal{P}(L)$ by a similar manner to that for computing projection of a finite automaton, i.e., (i) we first replace every unobservable event by ε and obtain a VPA with ε-transitions; (ii) next the VPA is transformed into one without ε-transitions by redefining the state transition function and by the determinizing procedure. The resulting VPA M' generates $\mathcal{P}(L)$ since occurrence of unobservable events does not change the stack as we have assumed.

A $\widehat{\Sigma}$-VPA $\mathcal{M}''$ with $L(\mathcal{M}'') = \mathcal{P}^{-1}(L)$ is obtained by simply adding self-loops with unobservable events to each state of $\mathcal{M}''$. ∎

As a consequence of the above lemma, we can show that for a finite automaton $\mathcal{M}_1$ over Σ_{int} and a $\widehat{\Sigma}$-VPA $\mathcal{M}_2$, $L(\mathcal{M}_1 \| \mathcal{M}_2)$ is a $\widehat{\Sigma}$-VPL. Therefore, we can compute $M \| P$ in Example 1 in the form of a VPA (Fig. 4), where broken arrows indicate transitions which violate P but are allowed in $G = M^{\uparrow\Sigma}$.

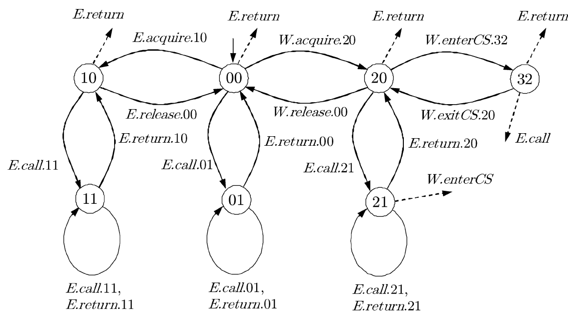

Fig. 5 $(M\|P)^{fin}$.

3.3 Finite-State Abstraction of VPAs

To make the problem tractable within a finite domain, we introduce the notion of *finite-state abstraction* of VPAs. Let $\mathcal{M} = (Q, \widehat{\Sigma}, \Gamma, \delta, q_0, Q_F)$ be a $\widehat{\Sigma}$-VPA. We define the finite-state abstraction of $\mathcal{M}$, denoted by $\mathcal{M}^{fin} = (Q, \Sigma.Q, \delta^{fin}, q_0, Q_F)$, as follows:

- $\Sigma.Q = \{\sigma.q \mid \sigma \in \Sigma, q \in Q\}$.
- $\delta^{fin}(q, \sigma.q') = q'$ if and only if one of the following three conditions holds:
    1. $\sigma \in \widehat{\Sigma}_{call}$ and $\delta_{call}(q, \sigma) = (q', \gamma)$ for some $\gamma \in \Gamma$;
    2. $\sigma \in \widehat{\Sigma}_{ret}$ and $\delta_{ret}(q, \sigma, \gamma) = q'$ for some $\gamma \in \Gamma$;
    3. $\sigma \in \widehat{\Sigma}_{int}$ and $\delta_{int}(q, \sigma) = q'$.

Note that adding the destination to the symbol on each transition is introduced to make the automaton deterministic. Moreover, we can extract the state information of VPA $\mathcal{M}$ from the sequences over $\Sigma.Q$.

Given a $\widehat{\Sigma}$-VPA $\mathcal{M}$, let $\mathcal{M}._{state}$ denote the VPA obtained by replacing the symbol $\sigma \in \Sigma$ on each transition with a symbol $\sigma.q \in \Sigma.Q$, where $q \in Q$ is the destination of the transition. Since conditions on the stack are ignored in the transitions of the finite-state abstraction, $L(\mathcal{M}._{state}) \subseteq L(\mathcal{M}^{fin})$ holds.

The finite-state abstraction of $M\|P$ is shown in Fig. 5.

4. Computation of Controllable and Normal Sublanguages

In this section, we show a method to compute a controllable and normal sublanguage based on finite-state abstraction of VPAs. The obtained language is not necessarily a supremal one, but the language is guaranteed to be controllable and normal.

4.1 Disabling Points

Let $K \subseteq L(G)$ be a prefix closed language. Instead of using K, we define the following language D_K:

$$D_K := \{s\sigma \mid s\sigma \in L(G) - K, s \in K\}.$$

Let $L_{D_K} := L(G) - D_K\Sigma^*$. Then $L_{D_K} = K$ holds. D_K corresponds to the set of points at which events are disabled in K w.r.t. $L(G)$. The idea of using D_K is similar to one in *control objectives* [12].

The iterative procedure $K_0 := K; K_{i+1} := K_i - ((L(G) - K_i)/\Sigma_{uc})\Sigma^*$ to compute $supC(K)$ is replaced with the following iterative procedure:

$$\begin{aligned} &D_0 := D_K; \\ &D_{i+1} := (D_i - \Delta_i) \cup \Delta_i/\Sigma_{uc}, \\ &\text{where } \Delta_i = D_i \cap \Sigma^*\Sigma_{uc} \end{aligned} \quad (1)$$

The procedure halts when $C_*(D_K) := D_{i+1} = D_i$.

Lemma 4.1: $L_{C_*(D_K)} = sup\, C(K)$.

Proof. Since $C_*(D_K)$ consists of sequences in the form $s\sigma$, $\sigma \in \Sigma_c$, $L_{C_*(D_K)}$ is controllable w.r.t. $L(G)$ and Σ_{uc}. Since $L_{C_*(D_K)} \subseteq K$, $L_{C_*(D_K)} \subseteq sup\, C(K)$ holds. Assume that there exists $s \in sup\, C(K) - L_{C_*(D_K)}$. Then there exists a prefix $s'\sigma$ of s such that $\sigma \in \Sigma_c$ and $s'\sigma \in C_*(D_K)$, i.e., all sequences of the form $s'\sigma\Sigma^*$ are disabled by $C_*(D_K)$. From the procedure of (1), there exists $s'' \in \Sigma_{uc}^*$ such that $s'\sigma s'' \in L(G) - K$. Since K is prefix-closed, so is $sup\, C(K)$, and therefore $s'\sigma \in sup\, C(K)$. This contradicts the fact that $sup\, C(K)$ is controllable. ■

The computation $supN(K) = K - \mathcal{P}^{-1}(\mathcal{P}(L(G) - K))\Sigma^*$ is replaced with the following computation on D_K:

$$N_*(D_K) := \mathcal{P}^{-1}(\mathcal{P}(D_K)) \quad (2)$$

Lemma 4.2: $L_{N_*(D_K)} = sup\, N(K)$.

Proof. We first show that $K' := L_{N_*(D_K)}$ is normal. Since $K' \subseteq L(G)$, $K' \subseteq \mathcal{P}^{-1}(\mathcal{P}(K')) \cap L(G)$ holds. Let $s \in K'$ and $s' \in L(G)$ such that $\mathcal{P}(s) = \mathcal{P}(s')$. In order to prove the other side $K' \supseteq \mathcal{P}^{-1}(\mathcal{P}(K')) \cap L(G)$, it suffices to show $s' \in K'$. Assume that $s' \notin K'$. Then $s' \in N_*(D_K)$ since $s' \in L(G)$, and therefore $\mathcal{P}^{-1}(\mathcal{P}(s')) \subseteq N_*(D_K)$ holds. This means that $s \in N_*(D_K)$. Contradiction. Hence, $L_{N_*(D_K)}$ is normal and $L_{N_*(D_K)} \subseteq sup\, N(K)$ holds.

Next we assume that there exists $s \in supN(K) - L_{N_*(D_K)}$. Since $s \in K$, there exists a prefix s' of s such that $s' \in N_*(D_K)$ and all sequences of the form $s'\Sigma^*$ are disabled by $N_*(D_K)$. Since $s' \notin D_K$ but $s' \in N_*(D_K)$, there exists $t \in D_K$ such that $\mathcal{P}(s') = \mathcal{P}(t)$. To make K normal, s', and also s, cannot be included in $sup\, N(K)$. Contradiction. ■

Thus, we obtain $L_{N_*(C_*(D_K))} = sup\, CN(K)$. The following lemma is immediately obtained from the fact that $sup\, CN(K) \subseteq K \subseteq L(G)$.

Lemma 4.3: $sup\, CN(K) = K - N_*(C_*(D_K))\Sigma^*$.

This lemma implies that when D_K is regular, $N_*(C_*(D_K))$ is computable even if K is non-regular. Suppose that $N_*(C_*(D_K))$ is computed as a finite automaton. Then $L_{N_*(C_*(D_K))} = K - N_*(C_*(D_K))\Sigma^*$ is computable if K is given as a VPA. This is because the class of VPLs are closed under difference, as described in 3.2. The obtained language $L_{N_*(C_*(D_K))}$ may be non-regular.

4.2 Approximating D_K

Let $K \subseteq L(G)$ be a prefix closed language. If D_K is non-regular, that we may consider a regular set $\widetilde{D}_K \subseteq \Sigma^*$ such that $D_K \subseteq \widetilde{D}_K$, and compute $N_*(C_*(\widetilde{D}_K))$. We call such $\widetilde{D}_K$ *an over approximation* of D_K. If K is given as a VPA, and consequently D_K is also a VPL, then we can use the finite-state abstraction of the VPA as an over approximation.

As we have seen in VPLs, approximating a non-regular language by a regular one is not a difficult task since most classes of automata for non-regular languages still have finite-state control, like in VPAs, pushdown automata, and also Turing machines. By ignoring infinite-state part of machines, we can obtain finite-state abstractions of them.

The following theorem shows that the resulting language is still controllable and normal even if we use an over approximation of D_K. As a result, we can have an assumption, which is not necessarily least restrictive, for any property given by a VPA. Note that approximating the language K does not give the controllability and the normality of the resulting language.

Lemma 4.4: Let $\widetilde{D}_K$ be an over approximation of D_K. Then $N_*(C_*(D_K)) \subseteq N_*(C_*(\widetilde{D}_K))$ holds.

Proof. Since $D_{i+1} = (D_i \cap \Sigma^*\Sigma_c) \cup (D_i \cap \Sigma^*\Sigma_{uc})/\Sigma_{uc}$, if $D_i \subseteq D'_i$ then $D_{i+1} \subseteq D'_{i+1}$ holds. Hence, $C_*(D_K) \subseteq C_*(\widetilde{D}_K)$. Moreover, if $D \subseteq D'$, then $\mathcal{P}(\mathcal{P}^{-1}(D)) \subseteq \mathcal{P}^{-1}(\mathcal{P}(D'))$ holds, and therefore $N_*(D) \subseteq N_*(D')$ holds. Thus, we obtain the result. ∎

Now we have the main result of this section.

Theorem 4.5: Let $\widetilde{D}_K$ be an over approximation of D_K, and let $\widetilde{L} := K - N_*(C_*(\widetilde{D}_K))$. Then $\widetilde{L}$ is controllable and normal. w.r.t. $L(G)$, Σ_c and Σ_o. Moreover, $\widetilde{L} \subseteq sup\ CN(K)$ holds.

Proof. Let $L_{\widetilde{D}_K} := L(G) - \widetilde{D}_K\Sigma^*$. Since $\widetilde{L} = sup\ CN(L_{\widetilde{D}_K})$ by Lemma 4.3, $\widetilde{L}$ is controllable and normal w.r.t. $L(G)$, Σ_c and Σ_o. Moreover, by Lemma 4.3 and Lemma 4.4, we have $\widetilde{L} \subseteq K - N_*(C_*(D_K)) = sup\ CN(K)$. ∎

4.3 Example

We go back to the instance of Problem *AGS* shown in Example 1. Let $\widetilde{D}_K$ be defined by $(M\|P)^{fin}$ shown in Fig. 5, i.e., $\widetilde{D}_K$ consists of all sequences in $(M\|P)^{fin}$ that end with events on broken arrows. Then $N_*(C_*(\widetilde{D}_K))$ is obtained as the finite automaton shown in Fig. 6, and the assumption is obtained as the VPA shown in Fig. 7, where the assumption A is computed by

$$L(A) = \ell\left(\mathcal{P}(L((M\|P)_{\cdot state}) - N_*(C_*(\widetilde{D}_K))\Sigma^*_{Q_{M\|P}})\right),$$

where $Q_{M\|P}$ is the set of states in $M\|P$, and ℓ is the labeling function that removes the state information from every event. The assumption is computable since VPLs are closed under difference, renaming, and also projection provided that all unobservable events are internal symbols, as shown in Sect. 3.2.

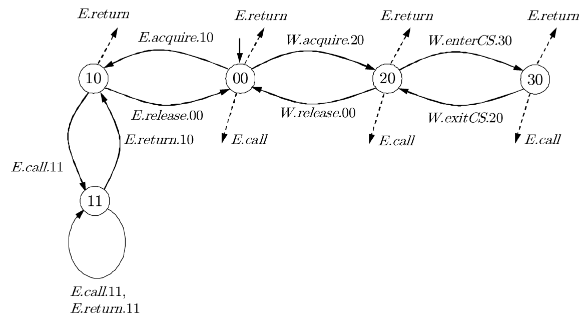

Fig. 6 $N_*(C_*(\widetilde{D}_K))$.

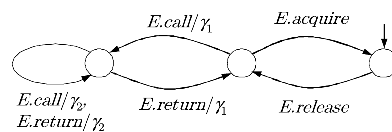

Fig. 7 The obtained assumption.

5. Assumption Generation for Liveness Properties

As a dual of Problem *AGS*, we can consider the most restrictive environment for which the software component satisfies given *liveness properties*, where a liveness property is a kind of properties like "*something good will happen*". *The assumption generation for liveness properties AGL* is the problem of finding an automaton A over $\mathcal{I} \cup (\Sigma_P - C)$ such that $L(P) \subseteq \mathcal{P}_{\Sigma_P}(L(M\|A))$. The automaton A describes the behavior of the environment that makes the composition $M\|A$ to satisfy liveness properties P. Contrary to the case for safety properties, we want have the strongest assumption. If there exists an environment E in which the component M satisfies the property, then all the behavior of M enabled by A are also enabled by E.

By the results on supervisory control theory, the strongest assumption exists, i.e., the infimal prefix-closed, controllable and normal sublanguage of a given language exists.

6. Conclusion

We have proposed a general scheme for computing the assumption even for non-regular properties, and shown the uniqueness of the lease restrictive assumption for any class of languages. Moreover, we have proposed a method to compute assumptions based on finite-state abstraction of automata. Since complements of languages are abstracted, the obtained assumption may still be non-regular.

The obtained result is restrictive since the software component M is required to be a finite automaton. Another possibility to the computability based on VPLs is to restrict

the operations so that the class of VPLs is closed under the operations necessary for computing the supremal controllable and normal sublanguages. We will show the results based on this idea in a separate reports. We remark that there exists a result on computing the supremal controllable sublanguage for non-regular specifications. In [15], it is shown that there is an algorithm to compute the supremal prefix-closed controllable sublanguage when (i) the plant is modeled by a finite automaton, and (ii) the prefix-closed specification language is generated by a deterministic pushdown automaton. Therefore, the difficulty lies in dealing with the projection operator for non-regular languages.

Dealing with non-regular properties in software component verification is necessary for modeling control structures and recursive calls in computer programs. If we restrict the depth of recursive calls or the number of iterations in loops, then the properties and the behavior of software components can be modeled by finite automata. However, this is quite inefficient. On the other hand, using context-free languages and the corresponding class of automata, pushdown automata, often falls into undecidability of verification problems. Using VPLs is a reasonable solution to the undecidability.

References

[1] E. Clarke, Jr., O. Grumberg, and D.A. Peled, Model Checking, The MIT Press, 1999.

[2] D. Giannakopoulou and C. Păsăreanu, "Assumption generation for software component verification," Proc. 17th IEEE Int. Conf. Automated Software Engineering, pp.3–12, 2002.

[3] C. Păsăreanu, M. Dwyer, and M. Huh, "Assume-guarantee model checking of software: A comparative case study," Lecture Notes in Computer Science, vol.1680, pp.168–183, 1999.

[4] R.J. Allen and D. Garlan, "A formal basis for architectural connection," ACM Trans. Softw. Eng. Methodol., vol.6, no.3, pp.213–249, 1997.

[5] D. Giannakopoulou, J. Kramer, and S.C. Cheung, "Behaviour analysis of distributed systems using the tracta approach," J. Automated Software Eng., vol.6, pp.7–35, 1999.

[6] C. Canal, E. Pimentel, and J.M. Troya, "Compatibility and inheritance in software architectures," Science of Computer Programming, vol.41, pp.105–138, 2001.

[7] F. Plasil and S. Visnovsky, "Behavior protocols for software components," IEEE Trans. Softw. Eng., vol.28, no.11, pp.1056–1076, 2002.

[8] D.M. Yellin and R.E. Strom, "Protocol specifications and component adaptors," ACM Trans. Programming Languages and Systems, vol.19, no.2, pp.292–333, 1997.

[9] C. de la Riva and J. Tuya, "Automatic generation of assumptions for modular verification of software specifications," J. Syst. Softw., vol.79, no.9, pp.1324–1340, 2006.

[10] R. Kumar and V.K. Garg, Modeling and Control of Logical Discrete Event Systems, Kluwer Academic Publishers, 1995.

[11] C.G. Cassandras and S. Lafortune, Introduction to Discrete Event Systems, Kluwer Academic Publishers, 1999.

[12] M.P. Spathopoulos and M.R. Laurence, "Supervisory control using control objectives," Proc. WODES'98, 1998.

[13] R. Alur and P. Madhusudan, "Visibly pushdown languages," Proc. 36th Annual ACM Symposium on Theory of Computing, pp.202–211, 2004.

[14] R. Alur, V. Kumar, P. Madhusudan, and M. Viswanathan, "Congruences for visibly pushdown languages," Proc. 32nd International Colloquium on Automata, Languages and Programming, LNCS 3580, pp.1102–1114, 2005.

[15] C. Griffin, "A note on the properties of the supremal controllable sublanguage in pushdown systems," IEEE Trans. Automatic Control, vol.53, no.3, pp.826–829, 2008.

Kunihiko Hiraishi received from the Tokyo Institute of Technology the B.E. degree in 1983, the M.E. degree in 1985, and D.E. degree in 1990. He is currently a professor at School of Information Science, Japan Advanced Institute of Science and Technology. His research interests include discrete event systems and formal verification. He is a member of the IEEE, IPSJ, and SICE.

Petr Kučera received from the Faculty of Mathematics and Physics, Charles University in Prague the master degree in 2001, and Ph.D. in 2005, both in theoretical computer science. He is currently an assistant professor at Department of Theoretical Computer Science and Mathematical Logic, Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic. His research interests include algorithms and Boolean functions.