Checking Subsystem Safety Properties in Compositional Reachability Analysis

Shing-chi Cheung
Department of Computer Science
Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong

Jeff Kramer
Department of Computing
Imperial College of Science, Technology and Medicine, London SW7 2BZ, UK

Email: scc@cs.ust.hk, jk@doc.ic.ac.uk
Fax: (+852) 2358-1477

Abstract

The software architecture of a distributed system can be described as a hierarchical composition of subsystems, with interacting processes as the leaves of the hierarchy. Process behaviour can be specified using finite-state machines. A global state machine describing the overall system behaviour can be constructed using compositional reachability analysis techniques. These techniques compose the global state machine of a system from its component processes in stages, based on the specified hierarchy. The key to the success of these analysis techniques is to employ a modular software architecture and hide as many internal actions as possible in each subsystem. A subsystem containing fewer observable actions can generally be represented by a simpler state machine. However, the properties that are available for reasoning (analysis) in the global state machine are constrained by the set of remaining globally observable actions. In this paper, we introduce a technique to check safety properties of subsystems which may contain actions that are not globally observable. We have found that these safety properties can still be checked in the framework of a compositional reachability analysis technique. Our technique is supported by augmenting the state machine model with a special state $\pi$. The state is used to capture possible violation of the safety properties specified by software developers. In the paper, safety properties are expressed using finite-state machines and concepts are illustrated using a gas station system as a case study.
1 Introduction

1.1 Background

Behaviour analysis is useful at all stages in the software life cycle. Its objectives are to discover software defects and to check if a system performs as intended. It is a crucial software engineering discipline for building quality into software. A distributed system is normally decomposed into an hierarchy of processes. The hierarchy, referred to as the compositional hierarchy, describes how the system can be built successively from its constituent processes. It reflects the conceptual view held by software developers of the design, and at the same time it offers the key to guide the compositional software analysis. Distributed systems are more complex to analyse than their sequential counterparts. The inherent nondeterminism dramatically increases the number of combinations of behaviour and generates a need to coordinate access to shared resources. Even for small systems, analysis of their behaviour is impractical without the support of an effective automated technique.

Each finite-state process in the compositional hierarchy can be modelled as a transition system (or state space graph) whose transitions are labelled by communicating actions. A formal definition of transition systems can be found in section 2. A process is primitive if a LTS describing its behaviour has been given explicitly; otherwise it is composite. For example, processes at the leaves of a compositional hierarchy are primitive, and processes representing subsystems are composite. In our discussion, we use subsystems and composite processes interchangeably and assume finite-state systems.

Static analysis techniques for concurrent systems can be used to verify two classes of system property: safety and liveness. A safety property asserts that the program never enters an undesirable state [1]. For example, mutual exclusion is a safety property which specifies the absence of a program state where a common resource is simultaneously accessed by more than one client. A liveness property asserts that a program eventually enters a desirable state [1]. For example, freedom from starvation is a liveness property; it says that a program state, where some request is served, will be finally entered.

In this paper, we focus our discussion on safety properties. Safety properties can be specified in terms of regular expressions or deterministic finite-state machines [7, 17]. The two formalisms are interchangeable. For the ease of understanding, we use the formalism of state machines. State machines that specify safety properties are called property automata. Each property automaton is so specified as to accept all execution sequences over the set of actions that correspond to a safety property of interest. For example, figure 1 gives a property
automaton of a safety property specifying that any execution of action *access* is to be preceded by an execution of *lock*.

### 1.2 Related Work

A common approach to the analysis of distributed systems is to construct a semantically equivalent representation of the global system. However, the search space involved generally increases dramatically with the number of parallel processes. Great effort has been made to avoid this state explosion problem by not having to construct the complete state graph. Roughly, the proposed methods can be classified into two categories: reduction by partial ordering and reduction by compositional minimisation. In the former category, reduction is achieved by avoiding the generation of all paths formed by the interleaving of the same set of transitions [8, 12, 23]. In the latter category, reduction is achieved by intermediate simplification of subsystems [14, 19-21, 25]. Techniques in this category are known as *compositional reachability analysis* (or CRA for brevity). They are originally proposed to remedy the problem of traditional reachability analysis techniques [2, 18, 22] which compose the global system representation in a single step. Promising results have been reported. Yeh [24] reported several case studies which suggested similar performance between a technique of compositional reachability analysis and that of constraint expressions [3]. Sahnani [19] reported an experiment applying compositional reachability analysis to the Q.931 protocol. They found that the intermediate state space graphs generated never exceeded 1,000 states although the global state space graph given by traditional reachability analysis of the protocol contained over 60,000 states.

Although CRA techniques have advantages over traditional techniques of reachability analysis, the system representation generated cannot be utilised to validate behavioural properties involving actions which are not globally observable. Verification is restricted to those properties formed only by globally observable actions. In this paper, we enhance the CRA techniques with a mechanism to validate safety properties of subsystems which contain actions that may not be globally observable. These properties are violated when those subsystems, within the context of a distributed application, can perform execution sequences not acceptable to the corresponding property automata. The validation can be carried out in the enhanced mechanism of CRA. If no violation of safety properties is detected, the analysis constructs a global LTS observationally equivalent [15] to that constructed using conventional CRA techniques; otherwise it indicates
which and how safety properties are violated. We have found no similar work of providing this feature in the framework of CRA. The proposed mechanism is adapted from the techniques of employing context constraints to alleviate the state explosion problem of CRA [4, 9]. Context constraints was originally proposed by Graf [9] to abstract behaviour restrictions imposed on a subsystem by its neighbouring processes due to the need for co-ordination. To enhance the mechanism of CRA, the state machine formalism is augmented with a special undefined state \( \pi \). The undefined state is used to capture potential violation of safety properties specified by users.

1.3 Paper Outline

In the next section, we introduce the labelled transition systems and present a gas station system which is to be used as a case study in our discussion. Section 3 presents a technique to detect and locate violation of safety properties related to subsystems. This is followed by conclusions in section 4.

2 Preliminaries

2.1 Labelled Transition System

Behaviour of a synchronous communicating process in a distributed system can be modelled as a labelled transition system (LTS for brevity). An LTS of a process is a state transition diagram containing all reachable states and executable transitions. For instance, figure 2 represents an LTS of a process \( \text{Proc} \) which can be in either state 0 or 1. The process can go from state 0 to 1 as the consequence of an action \( \text{prepay1} \) and back to state 0 from 1 by an action \( \text{charge1} \). Alternatively, the process can go back to state 0 from 0 by executing \( \text{charge2} \). Note that property automata are also LTSs.

The set of actions that are considered relevant for a particular description of a process \( P \) is called its communicating alphabet, written as \( \alpha P \). In the above example, \( \alpha \text{Proc} \) equals \{\text{prepay1, charge1, charge2}\}. The alphabet is a permanent predefined property of a process. It is logically impossible for a process to perform an action outside its alphabet. The choice of an alphabet is essentially a deliberate simplification to make analysis practical. This simplification involves decisions to ignore many other properties and actions considered to be of lesser
interest.

Formally, an LTS of a process $P$ is a quadruple $< S, A, \Delta, q >$ where

(i) $S$ is a set of states;

(ii) $A = A' \cup \{ \tau \}$, where $A'$ is the communicating alphabet of $P$ which does not contain the internal action $\tau$;

(iii) $\Delta \subseteq S \times A \times S$, denotes a transition relation that maps from a state and an action onto another state;

(iv) $q$ is a state in $S$ which indicates the initial state of $P$.

An LTS of $P = < S, A, \Delta, q >$ transits into another LTS of $P' = < S, A, \Delta, q' >$ with an action $a \in A$ if and only if $(q, a, q') \in \Delta$ and $q' \neq \pi$, where $\pi$ is a special undefined state, discussed further below. That is,

$\langle S, A, \rightarrow, q \rangle \xrightarrow{a} \langle S, A, \rightarrow, q' \rangle$ iff $(q, a, q') \in \Delta$ and $q' \neq \pi$.

Since there is an one-to-one mapping between a process $P$ and its LTS, we use the term process and LTS interchangeably. Therefore, the above statement can be rewritten as follows:

$P \xrightarrow{a} P'$ iff $(q, a, q') \in \Delta$ and $q' \neq \pi$.

An LTS $< S, A \cup \{ \tau \}, \Delta, q >$ is said to be deterministic if and only if

$\forall s, s' \text{ and } s'' \in S,$

$(s, a, s') \in \Delta \land (s, a, s'') \in \Delta$ implies $s' = s''$;

otherwise it is said to be non-deterministic.

A trace of a process $P$ is a sequence of actions that $P$ can perform starting from its initial state. For example, the sequence $< \text{ prepay1, charge1, charge2 }>$ is a trace of Proc in figure 2. We denote the set of possible traces of a process $P$ as $\text{tr}(P)$. The formal definition of traces can be found in the work of Hoare [11]. An LTS may contain a special state $\pi$, which is called the undefined state. A process is considered to be undefined when it is in state $\pi$. An undefined process is denoted as $\Pi = < \{ \pi \}, A, \emptyset, \pi >$, where $A$ is the universal set of actions (see figure 3(a)). A process $P = < S, A, \Delta, q >$ transits into $\Pi$ after the execution of an action $a$ if and only if $P$ can perform the transition $(q, a, \pi)$. That is,

$\langle S, A, \rightarrow, q \rangle \xrightarrow{a} < \{ \pi \}, A, \emptyset, \pi >$ iff $(q, a, \pi) \in \Delta$.

Alternatively, we may write:

$P \xrightarrow{a} \Pi$ iff $(q, a, \pi) \in \Delta$. 

6
For instance, figure 3(b) represents a process \( \text{Proc} \) that transits into \( \Pi \) after executing \textit{charge1}. Hence, \( P^{\text{charge1}} \rightarrow \Pi \).

In the LTS computational model, there can be no transitions emerging from state \( \pi \). Thus, \( \Pi \) is essentially a termination process. A trace is said to be undefined if the execution turns a process into \( \Pi \); otherwise it is said to be defined. For instance, \( \langle \text{charge2}, \text{charge1} \rangle \) is an undefined trace in \( \text{Proc} \). Normally, all processes in a system design are defined; they are free of undefined state \( \pi \) and do not contain undefined traces. The constructs of \( \pi \) and \( \Pi \) are used to detect violation of safety properties specified by users.

In the LTS computational model, actions of a process can be restricted by a restriction operator \( \uparrow \). \( P \uparrow L \) represents the process projected from \( P \) in which only the actions in set \( L \) are observable. The restriction operator ensures that \( P \) has undefined traces if and only if \( P \uparrow L \) has undefined traces.

\[
\begin{align*}
1a. \quad & P \xrightarrow{a} P' \\
& P \uparrow L \xrightarrow{a} P' \uparrow L \quad (a \in L, P' \neq \Pi) \\
1b. \quad & P \xrightarrow{a} \Pi \\
& P \uparrow L \xrightarrow{a} \Pi \quad (a \in L) \\
2a. \quad & P \xrightarrow{a} P' \\
& P \uparrow L \xrightarrow{\tau} P' \uparrow L \quad (a \in L, P' \neq \Pi) \\
2b. \quad & P \xrightarrow{a} \Pi \\
& P \uparrow L \xrightarrow{\tau} \Pi \quad (a \in L)
\end{align*}
\]

Processes in a distributed system can also be composed by a composition operator \( \parallel \) similar to that used in CSP [11]. \( P \parallel Q \) is the parallel composition of processes \( P \) and \( Q \) with synchronisation of the actions common to both of their alphabets and interleaving of the others. The alphabet of \( P \parallel Q \) is given by the union of their individual alphabets (i.e. \( \alpha P \cup \alpha Q \)).

The transition relation for the composite process is defined by rules 3, 4 and 5. The composition operator is both commutative and associative. Figure 4 shows the LTS of \( A \parallel B \) composed from processes \( A \) and \( B \). A composite process \( P \parallel Q \) never proceeds if either \( P \) or \( Q \)
equals the undefined process $\Pi$ whose alphabet is the universal set of actions. The rules enforces that a composite process $P \parallel Q$ is undefined when $P$ or $Q$ is undefined.

3a. $\frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q}$ \hspace{1cm} (a \notin \alpha Q, P' \neq \Pi)

3b. $\frac{P \xrightarrow{a} \Pi}{P \parallel Q \xrightarrow{a} \Pi}$ \hspace{1cm} (a \notin \alpha Q)

4a. $\frac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'}$ \hspace{1cm} (a \notin \alpha P, Q' \neq \Pi)

4b. $\frac{Q \xrightarrow{a} \Pi}{P \parallel Q \xrightarrow{a} \Pi}$ \hspace{1cm} (a \notin \alpha P)

5a. $\frac{P \xrightarrow{a} P' \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P' \parallel Q'}$ \hspace{1cm} (a \in \alpha P \cap \alpha Q, P' \neq \Pi, Q' \neq \Pi)

5b. $\frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} \Pi}$ \hspace{1cm} (a \in \alpha P \cap \alpha Q, P' = \Pi or Q' = \Pi)

The restriction operator $\uparrow$ can be distributed into the composition operator $\parallel$ if internal actions of constituent processes do not have conflicting names. That is,

6. for any two processes $P, Q$ and two sets of actions $L, M$:

$$(\alpha P - L) \cap (\alpha Q - M) = \emptyset$$

implies $P \uparrow L \parallel Q \uparrow M = (P \parallel Q) \uparrow (L \cup M)$

The sufficient condition in rule (6), which can be achieved by action renaming, is assumed in our discussion.

2.2 : Behavioural Equivalences

Strong semantic equivalence, denoted as $\sim$, is used to relate two processes whose behaviours are indistinguishable to an observer even if internal $\tau$-actions are observable. Weak semantic equivalence, denoted as $\approx$, is used to relate two processes whose behaviours are indistinguishable to an observer if internal $\tau$-actions are not observable. Both equivalences assume external observers are notified if processes have assumed state $\pi$. Let $A$ be the basic set of actions including $\tau$. A strong semantic equivalence $\sim$ is the union of all relations $R \subseteq S \times S$ satisfying that $(P, Q) \in R$ implies:

(i) $P = \Pi$ if and only if $Q = \Pi$.

(ii) for all $a \in A$:

(a) $P \xrightarrow{a} P'$ implies $\exists Q', Q \xrightarrow{a} Q'$ and $(P', Q') \in R$.

(b) $Q \xrightarrow{a} Q'$ implies $\exists P', P \xrightarrow{a} P'$ and $(P', Q') \in R$.

Let $P \xrightarrow{\tau} P'$ denote $P(\tau)^* \xrightarrow{a} (\tau)^* P'$. A weak semantic equivalence $\approx$ is the union of all relations $R \subseteq S \times S$ satisfying that $(P, Q) \in R$ implies:

(i) $P = \Pi$ if and only if $Q = \Pi$.

(ii) for all $a \in A$:
Figure 5: Behaviour of the Operator and the Queue holding customers' requests.

(a) \( P \xrightarrow{a} P' \) implies \( \exists Q', Q \xrightarrow{a} Q' \) and \( (P', Q') \in R \).
(b) \( Q \xrightarrow{a} Q' \) implies \( \exists P', P \xrightarrow{a} P' \) and \( (P', Q') \in R \).

The strong and weak semantic equivalences coincide with the strong and weak equivalence (cf. Milner [16]) respectively if the first of the two requirements (i) is dropped. In other words, if two processes \( P \) and \( Q \) are free of the undefined state \( \pi \) then

(i) \( P \sim Q \) implies \( P \) and \( Q \) are strongly equivalent;
(ii) \( P = Q \) implies \( P \) and \( Q \) are weakly equivalent.

2.3 A Gas Station Example

As an illustration of our discussion, we present a gas station example originally proposed by Helmbold and Luckham [10]. The system models an automated gas station with an operator, a pump, two customers and a queue holding customers' requests. Figure 5 gives the LTSs presenting the behaviour of the operator and the request queue.

The operator may initially choose to accept money prepaid by customers (\( \text{prepay}_i \)) or accept the amount to be charged from the pump (\( \text{charge}_i \)). After accepting money from a customer, the operator activates the pump if it is available; otherwise does nothing. On receiving the charge information from the pump, the operator gives the change (\( = \text{prepay}_i - \text{charge}_i \)) to the customer and activates the pump again if there are other customers waiting for the pump. Figure 6 shows the behaviour of the pump and that of the two customers. A customer who has paid the money can start the pump once it has been activated. After starting the pump, the customer may at any time request the pump to finish pumping and wait for the change from the operator. Upon receiving the "finish" request, the pump informs the operator of the charge information.

The gas station system\(^1\) is formed by a parallel composition of the primitive processes

\(^{1}\) If necessary, the system can be extended to accommodate more customers and pumps.
Figure 6: Behaviour of the Pump and two customers Cust1 and Cust2

Figure 7: Compositional Hierarchy of the Gas Station System

Operator, Queue, Pump, Cust1 and Cust2 using a compositional hierarchy as shown in figure 7. Subsystems, represented by boxes with rounded corners, are composite processes formed by composition of simpler subsystems or primitive processes. Primitive processes, represented by boxes with sharp corners, are leaves of the compositional hierarchy. The behaviours of primitive processes are given as LTSs.

The hierarchy reflects a conceptual view held by software developers of the gas station system. Subsystems are introduced to make the system more modular and comprehensible. This is achieved by hiding internal actions of a subsystem from external processes. For instance, subsystem Counter hides actions pump_avail, pump_occupied, cust_none and cust_wait from processes outside the Counter. Textually, we write:

\[
\text{Counter} = (\text{Operator} \parallel \text{Queue}) \uparrow L, \text{ where} \\
L = (\alpha\text{Operator} \cup\alpha\text{Queue}) - \{\text{pump_avail, pump_occupied, cust_none, cust_wait}\}
\]

Let us assume in the following discussion that the software developers wish to reason about the global behaviour of the system on actions prepay1 and prepay2. In other words, only these two actions are observable in the global LTS of GasSystem.

\[
\text{GasSystem} = (\text{Station} \parallel \text{Clients}) \uparrow \{\text{prepay1, prepay2}\}
\]
3. Enhanced Compositional Reachability Analysis

3.1 Limitations of Compositional Reachability Analysis

Recent literature has reported promising results by employing a compositional approach to generate the model of a system using reachability analysis [19, 20, 25]. In compositional reachability analysis techniques, the model of the target system is given as an LTS which describes the overall system behaviour. Given a compositional hierarchy, the LTS of a system is composed step by step from those of its subsystems in a bottom-up manner. In each intermediate step, the LTS of a subsystem is simplified by hiding all internal actions. For instance, the LTS of GasStation in figure 7 can be composed in four steps. First, compose the LTS of Counter from Operator and Queue. Second, use that LTS and Pump to generate the LTS of Station. Third, construct the LTS of Clients from Cust1 and Cust2. Fourth, compose the LTS of GasStation from that of Station and that of Clients. This mechanism of "intermediate simplification during composition" is attractive for the analysis of modular systems. Figure 8 shows the global LTS thus obtained. It shows that the gas system repeatedly accepts money from Cust1 and Cust2 by means of actions prepay1 and prepay2. There is no particular ordering relation between the occurrences of prepay1 and prepay2.

The key to the success of CRA techniques is to employ a modular software architecture and hide as many internal actions as possible in each subsystem. A subsystem containing fewer observable actions can generally be represented by a simpler LTS. However, the properties that are then available for reasoning in the analysis is constrained by the set of remaining globally observable actions. For instance, the properties that are available for reasoning in the analysis of the GasStation can only be formed by actions prepay1 and prepay2. Safety properties of subsystems that involve other actions cannot be examined from the global LTS of the GasStation in figure 8. Examples of these other properties are that the Operator must give the right change to the right customer (figure 9(a)) and the Pump must complete the service to a customer before serving the other (figure 9(b)). If these properties are to be verified in the CRA, actions charge1, charge2, change1, change2, start1, start2, finish1 and finish2 need to be made globally observable. However, this would go against the key hiding principle of CRA techniques and thus undermine the effectiveness of the associated analysis. In the following, we introduce a technique that is capable of checking these safety properties without increasing the set of globally observable actions in the GasStation.
3.2 Validation of Safety Properties

Let $P$ be a process equals $Q \uparrow L$. A safety property of a process $P$ can be represented by a deterministic property automaton $T = (A, S, \Delta, q)$ where
- $T$ is free of undefined traces and internal action $\tau$; and
- $\alpha T \subseteq \alpha Q$.

For example, the property automaton Right Change in figure 9(a) can be used to specify a safety property of Operator, Counter or Station in figure 7. The property is said to be violated by a process, within the context of a distributed system, if and only if it can perform a trace not acceptable to the property automaton. In other words, a safety property holds for a process if and only if it holds for the whole system. Let $Z = Operator \parallel Queue \parallel Pump \parallel Cust1 \parallel Cust2; GasStation$ behaves the same as $Z \uparrow \{prepay1, prepay2\}$. The property represented by Right Change holds for Operator if and only if:

$$tr( Z \uparrow \alpha Right Change ) \subseteq tr( Right Change ).$$

The essence of violation detection is to derive an image automaton based on a given property automaton. This image automaton is then introduced into the system so that the automaton is forced into an undefined state when the system performs a trace not acceptable to the original property automaton. The fact that the image automaton can attain state $\pi$ can be confirmed by the existence of a reachable state $\pi$ in the global LTS. The image property automaton $T'' = (A, S \cup \{\pi\}, \Delta', q)$ of a given property automaton $T = (A, S, \Delta, q)$ can be derived using the following two steps:

---

2 The internal action $\tau$ is not useful for specifying safety properties.
1. initialise $\Delta'$ to $\Delta$
2. for all $a \in A$ and $s \in S$ where there does not exist $s' \in S$ such that $(s, a, s') \in \Delta$:
   add $(s, a, \pi)$ to $\Delta'$

For example, figure 10 gives the corresponding image automata for Right Change and Right Service of figure 9.

The image automaton so constructed satisfies two conditions:
(i) $T$ and $T'$ have the same set of defined traces (i.e. $\text{tr}(T)$); and
(ii) for any process $P$,
   $P \parallel T'$ does not contain undefined traces if and only if $\text{tr}(P \hat{\to} \alpha T') \subseteq \text{tr}(T)$.

Proof of condition (i)
Step (1) in the construction of image automaton ensures that $\Delta$ is a subset of $\Delta'$. Step (2) ensures that for any transition $(s, a, s')$ belongs to $\Delta' \Delta$, $s'$ equals $\pi$. Hence, the $\Delta'$ and $\Delta$ contain the same set of transitions that do not involve state $\pi$; i.e.

$\{(s, a, s') | (s, a, s') \epsilon \Delta' \wedge s' \neq \pi\} = \{(s, a, s') | (s, a, s') \epsilon \Delta \wedge s' \neq \pi\}$.

Since $T$ and $T'$ share the same initial state $q$, all defined traces that can be performed by $T$ can also be performed by $T'$ and vice versa. As a result, $T$ and $T'$ have the same set of defined traces, which is equal to $\text{tr}(T)$. $\square$

Proof of condition (ii)
(a) if part: Suppose $\text{tr}(P \hat{\to} \alpha T') \subseteq \text{tr}(T)$. The set of traces $\text{tr}((P \parallel T') \hat{\to} \alpha T') = \{ t | t \in \text{tr}(P \hat{\to} \alpha T') \cap \text{tr}(T') \}$. The supposition implies that all traces in $\text{tr}(P \hat{\to} \alpha T')$ are defined. As a result, all traces in $\text{tr}((P \parallel T') \hat{\to} \alpha T')$ are defined. By the semantics of restriction operator, absence of undefined traces in $(P \parallel T') \hat{\to} \alpha T'$ implies absence of undefined traces in $P \parallel T'$.

(b) only-if part: Suppose $P \hat{\to} \alpha T$ can perform a trace $t$ that does not belong to $\text{tr}(T)$. The supposition implies that any prefix of $t$ can also be performed by $P \hat{\to} \alpha T'$. Let $s$ be a prefix of $t$ such that $T$ can perform all prefixes of $s$ but not the $s$ itself. By step (2) in the image automaton construction, $s$ can also be performed by $T'$. As a result, $s$ is an undefined trace in $T'$. By rules (3,4b) of the composition operator, $s$ is an undefined trace in $(P \hat{\to} \alpha T' \parallel T')$, which is equal to $(P \parallel T') \hat{\to} \alpha T'$. By the semantics of restriction operator, the existence of an undefined trace in $(P \parallel T') \hat{\to} \alpha T'$ implies the existence of an undefined trace in $P \parallel T'$. Hence the supposition cannot hold if $P \parallel T'$ does not contain any undefined traces. $\square$

Condition (ii) enables us to detect violation of safety properties in a system by checking the
existence of undefined traces in the composite process formed by the system and the image property automata. If undefined traces exist in the composite process, some safety properties are violated. An image automaton can be composed directly with a component process, whose alphabet is a superset of that of the automaton, in the CRA. For example, figure 11 shows the modified compositional hierarchy to include the image property automata of figure 10. Figure 12 gives the global LTS derived by the CRA based on the hierarchy in figure 11. Since the global LTS contains undefined traces, the Gas Station system does not satisfy both safety properties represented by the property automata Right_Change and Right_Service.

3.3. Locating the Violation

The above technique gives the information whether all specified safety properties are satisfied. However, users would normally wish to know which particular safety properties are violated, and in what way. To provide this information, the CRA technique can be further enhanced with a mechanism to keep track of the relation between those transitions leading to the undefined state $\pi$ in the global LTS and those in the image property automata.

For convenience, we use the notation $[P \xrightarrow{a} \Pi]$ to represent the set of transitions in the image property automata that contributed to the transition of $P \xrightarrow{a} \Pi$. We call $[P \xrightarrow{a} \Pi]$ the set of ancestor transitions of $P \xrightarrow{a} \Pi$. For instance, let Right_Change'$_0$ be Right_Change' in figure 10(a). Since $(0, changeI, \pi)_{Right_Charge}$ is the only transition that contributes the transition of $Right_Change'_{0} \xrightarrow{changeI} \Pi$, the value of $[Right_Change'_{0} \xrightarrow{changeI} \Pi]$ is therefore given by
{(0, change1, π)_{Right\_Charge'}}. The subscript Right\_Charge' identifies the image automaton which owns the transition. Similarly let Right\_Change'_0^{\text{charge1}} \rightarrow \text{Right\_Change}'_1 and the value of \([\text{Right\_Change}'_1^{\text{charge1}} \rightarrow \Pi]\) is given by \{(1, change1, π)_{Right\_Charge'}\}. In the CRA technique, processes are successively composed, restricted to some actions and then simplified based on the rules in section 2. In each of these analysis steps, the set of ancestor transitions are updated by following rules:

(a) In the step of parallel composition using rules (3b, 4b, 5b), update

- \([P\ll Q \xrightarrow{a} \Pi]\) to \([P \xrightarrow{a} \Pi]\) if \(Q \xrightarrow{a} \Pi\)
- \([P \ll Q \xrightarrow{a} \Pi]\) to \([Q \xrightarrow{a} \Pi]\) if \(P \xrightarrow{a} \Pi\)
- \([P \ll Q \xrightarrow{a} \Pi]\) to \([P \xrightarrow{a} \Pi] \cup [Q \xrightarrow{a} \Pi]\) if \(P \xrightarrow{a} \Pi\) and \(Q \xrightarrow{a} \Pi\)

(b) In the step of action restriction, update

- \([P \uparrow L \xrightarrow{a} \Pi]\) to \([P \xrightarrow{a} \Pi]\) if \(P \uparrow L \xrightarrow{a} \Pi\) is derived using rule (1b)
- \([P \uparrow L \xrightarrow{\tau} \Pi]\) to \([P \xrightarrow{a} \Pi]\) if \(P \uparrow L \xrightarrow{\tau} \Pi\) is derived using rule (2b)

(c) In the step of process minimisation, update

both \([P \xrightarrow{a} \Pi]\) and \([Q \xrightarrow{a} \Pi]\) to \([P \xrightarrow{a} \Pi] \cup [Q \xrightarrow{a} \Pi]\) if \(P = Q\).

In figure 12, let \textit{GasStation}_0 be \textit{GasStation}, \textit{GasStation}_0 \xrightarrow{\text{prepay1}} \textit{GasStation}_1 and \textit{GasStation}_1 \xrightarrow{\tau} \textit{GasStation}_2. Using the above updated rules of ancestor sets, the value of \([\text{GasStation}_2 \xrightarrow{\tau} \Pi]\) is given by \{(1, change2, π)_{Right\_Charge'}, (2, change1, π)_{Right\_Charge'}\}. The subscript of the tuples in \([\text{GasStation}_2 \xrightarrow{\tau} \Pi]\) indicates that the violation appears in the safety property specified by Right\_Change. The tuples suggest that violation is contributed by the transitions Right\_Change'_1^{\text{change2}} \rightarrow \Pi and Right\_Change'_2^{\text{change1}} \rightarrow \Pi in figure 10(a). The former represents the situation where change1 is followed by change2 and the latter represents the situation where change2 is followed by change1. In either situation, customers receive the
Suppose $Z$ and $Ifc$ are two processes; and $\sim$ denotes the strong semantic equivalence relation.

$$Z \sim (Z \parallel Ifc)$$

if

(i) $\alpha Ifc \subseteq \alpha Z$;

(ii) $tr(Z \uparrow \alpha Ifc) \subseteq tr(Ifc)$;

(iii) $Ifc$ is a deterministic process free of internal action $\tau$.

Figure 14: Interface Theorem

wrong change. The violation can be corrected by replacing the $Pump$ in figure 6 by that in figure 13(a). The global LTS thus constructed based on the hierarchy in figure 11 is given in figure 13(b). As the global LTS does not contain any reachable undefined state $\pi$, and hence there is no violation of safety properties represented by $Right\_Change$ and $Right\_Service$.

A prototype based on the concepts discussed has been implemented on a Sun Sparc/IPX workstation. The prototype required 1.11 seconds to compute the global LTS of $GasStation$ based on the compositional hierarchy in figure 7 and process specification in figures 5 and 6. When image automata $Right\_Change'$ and $Right\_Service'$ were included in the compositional hierarchy in figure 11, the prototype computed the global LTS given by figure 12 in 0.97 seconds. When $Pump$ was re-specified as the LTS in figure 13(a), the computation time became 1.09 and 1.02 seconds with and without inclusion of the image automata.

3.4 Correctness of the Global LTS

In the previous work of the authors, it has been shown that the overall behaviour of a system $Z$ remains unchanged after the addition of a process $Ifc$ if $Z$ and $Ifc$ satisfy the three criteria in an interface theorem (figure 14). Let $Z \uparrow L$ be a target system and $Ifc$ be an image property automaton specified by users. Using the construction mechanism for image automata ensures that $Ifc$ and $Z$ satisfy criteria (i) and (iii) in the theorem. In addition, it also ensures that $Z \parallel Ifc$ does not contain undefined traces if and only if $tr(Z \uparrow \alpha Ifc) \subseteq tr(Ifc)$. As a result, the absence of reachable undefined state $\pi$ in the global LTS of $Z \parallel Ifc$ implies $tr(Z \uparrow \alpha Ifc) \subseteq tr(Ifc)$, the satisfaction of criteria (ii) in the theorem. Thus a global LTS derived with the inclusion of $Ifc$
represents the overall behaviour of $Z$ if the LTS is free from undefined traces\(^3\). For example, figure 13(b) gives the global LTS constructed with the inclusion of `Right Change' and `Right Service'. Since the LTS does not contain any undefined traces, it represents the overall behaviour of `GasStation'.

4. Conclusion

The paper presents a mechanism to check safety properties associated with subsystems in the framework of CRA techniques. These safety properties are specified in terms of deterministic finite-state machines called property automata which may involve actions that are not globally observable. The property automata are said to be violated if the associated subsystems can perform traces not acceptable to them. An image automaton can be derived from each given property automaton. The image automaton is trapped into a special undefined state $\pi$ when the associated subsystem performs a trace which is not acceptable to the original property automaton. This can be identified directly from the existence of a reachable state $\pi$ in the global LTS. If the LTS is free from state $\pi$, it represents the overall behaviour of the system; otherwise the mechanism indicates which safety properties are violated and how they happen. The mechanism can be further optimised by augmenting the CRA technique with the concept of context constraints [4]. These constraints capture behavioural restriction imposed on subsystems by their neighbouring processes.

A prototype supporting the technique has been built. To further explore the potential of the technique, we are applying it to more complex examples, implementing support tools on workstations, and proposing to incorporate this form of analysis support in an environment for the design and construction of distributed systems, the System Architect's Assistant [13]. We also developing a framework to integrate this enhanced CRA technique with a dataflow analysis technique [5, 6].

References


\(^3\) Strong semantic equivalence is a subrelation of weak semantic equivalence. In the situations where there is an absence of state $\pi$, the strong and weak semantic equivalence coincide with the strong and weak equivalence respectively as defined by Milner [15].


