Checking and Correcting Safety Properties using 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

Abstract

The software architecture of a distributed program can be represented by a hierarchical composition of subsystems, with interacting processes at the leaves of the hierarchy. Compositional reachability analysis (CRA) is a promising state reduction technique which can be automated and used in stages to derive the overall behaviour of a distributed program based on its architecture. CRA is particularly suitable for the analysis of programs which are subject to evolutionary change. When a program evolves, only the behaviours of those subsystems affected by the change need be re-evaluated. The technique however has a limitation. The properties available for analysis are constrained by the set of actions that remain globally observable. Properties involving actions encapsulated by subsystems may therefore not be analyzed. In this paper, we enhance the CRA technique to check safety properties which may contain actions that are not globally observable. To achieve this, the state machine model is augmented with a special trap state labelled as $\pi$. We propose a scheme to transform in stages a property that involves hidden actions to one that involves only globally observable actions. The enhanced technique also includes a mechanism aiming at reducing the debugging effort. The technique is illustrated using a case study of a gas station system.
1. Introduction

1.1 Background

Distributed processing is widely used to provide computing support for many diverse applications. Many of these applications are complex and critical; an error can have catastrophic consequences. Behaviour analysis is one of the techniques that can help to discover defects and check if a program performs as intended. However, concurrent and distributed programs are generally more complex to analyze than their sequential counterparts. Even for small programs, analysis of their behaviour is impractical without the support of an effective automated technique.

Static analysis techniques for concurrent and distributed programs can be used to verify two classes of property: safety and liveness [1]. A safety property asserts that the program never enters an undesirable state. 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. For example, freedom from starvation is a liveness property; it says that a program state, where some request is served, will finally be entered.

In this paper, we focus our discussion on the provision of an effective technique for checking safety properties. Safety properties can be specified in terms of regular expressions or deterministic finite-state machines [11, 27]. 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 specifies the set of feasible execution sequences over the actions (transitions) that correspond to a safety property of interest. For example, the property automaton in Figure 1 asserts that no execution of action update can occur unless preceded by an execution of action lock.

![Image of a property automaton](image-url)  

**Figure 1: A Property Automaton**
The software architecture of a distributed program can be represented by an hierarchical composition of subsystems, with interacting (primitive) processes at the leaves of the hierarchy. Behaviour of a primitive process can similarly be modelled as a state machine whose transitions are labelled by the activities it can perform. Composite processes appear at the nodes of the hierarchy. Each composite process is a subsystem formed by a collection of processes; these processes can be either primitive or composite.

Analysis of such distributed programs is conducted by constructing an 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 [15, 19, 34]. In the latter category, reduction is achieved by intermediate simplification of subsystems [24, 30, 31, 32, 37]. Techniques in this category are known as compositional reachability analysis (CRA). We adopt this latter approach as it is effective, amenable to automation and complements and mirrors the software architecture of the program design.

1.2 Related Work

CRA techniques were originally proposed to remedy the problem of traditional reachability analysis techniques [2, 28, 33] which compose the global system representation in a single step. Yeh [36] described several case studies which suggested similar performance between a technique of compositional reachability analysis and that of constraint expressions [3]. Sabnani [30] described 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. A similar observation has also been made by [13] in the case study of a reliable multicast transport protocol that contains over 96,000 states and 672,000 transitions. The intermediate state space graphs generated by the CRA never exceeded 400 states. Furthermore, CRA techniques are particularly suitable to analyse programs which are likely to evolve. The techniques help localize the effect of change. When changes are applied to a program, only the state machines of those subsystems that are affected by the changes need be re-computed.
Although CRA techniques have advantages over traditional techniques of reachability analysis, the system representation generated cannot be utilised to validate behavioural properties involving actions that are not globally observable. Verification has to be restricted to only those properties formed by globally observable actions. However, a complex program typically involves many sophisticated interactions between multiple subsystems. These subsystems can be independently developed or extracted from software libraries. Each assumes a set of predefined communicating protocols at its interface. For a subsystem to function properly, its protocols must be respected by its neighbours. These protocols can therefore be conceived as the subsystem properties that have to be satisfied in the composite system. Checking the satisfaction of all these properties in the global context may therefore lead to the need to expose all subsystem interactions even though many of them may be irrelevant to the abstract view of the system. This contradicts the key philosophy of abstraction and jeopardizes the effectiveness of CRA techniques when applied to complex programs.

In this paper, we enhance the CRA technique with a mechanism to validate safety properties without the need for making the involved actions globally observable. These properties are violated when some subsystems, within the context of a distributed application, can perform execution sequences not acceptable to the specified 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 [25] to that constructed using conventional CRA techniques; otherwise it indicates which and how safety properties are violated. We have found no similar work providing this feature in the framework of CRA. The proposed mechanism is adapted from the state space reduction techniques for CRA [9, 16]. To check safety properties in the mechanism of CRA, we add to the state machine model a special trap state. The trap state, labelled as $\pi$, is used to capture potential violation of safety properties specified by users. The same philosophy has also been utilised to detect erroneous context constraints [7].

1.3 Paper Outline

In the next section, we introduce labelled transition systems and present a gas station system which is used as a case study in our discussions. Section 3 presents a technique to detect and locate violation of safety properties in the framework of CRA, illustrating experience of its use on the case study. Section 4 gives an account of a mechanism which reduces the effort of debugging in the compositional framework. This is followed by performance evaluation and conclusions in sections 5 and 6, respectively.
2. Background

2.1 Labelled Transition Systems

A labelled transition system (LTS) can be used to model the behaviour of a synchronously communicating process in a distributed program. An LTS contains all the states that the process may reach and all the transitions it may perform. For instance, Figure 2 represents an LTS describing a writer that repeatedly updates information in a shared resource. The Writer can go from state 0 to 1 as the consequence of locking the shared resource. Unlocking the resource triggers the opposite transition.

Similar activities performed by a program are labelled by the same action if there is no need to distinguish them from one another. The set of actions which are considered relevant for a particular description of a process \( P \) is called its alphabet [18]. In the above example, the alphabet of \( \text{Writer} \) equals \{lock, unlock, update\}. The alphabet is a permanent predefined property of a process. Logically, a process may only perform actions belonging to its alphabet. For example, \( \text{Writer} \) cannot perform an action \( \text{read} \) which is outside its alphabet. However, a process might never perform an action in its alphabet. An alphabet is so chosen as to make analysis tractable. This involves decisions to ignore many other properties and actions considered to be of lesser interest. A process may perform some activities which cannot be influenced by its environment. These activities are labelled by an internal action which is represented by the symbol \( \tau \). The presented LTS computational model has been widely used in the literature for specifying and analysing distributed programs [10, 12, 20, 29, 35]. Formally, an LTS of a process \( P \) is a quadruple \(< S, A, \Delta, q >\), where

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

(ii) \( A = \alpha P \cup \{\tau\} \), where \( \alpha P \) denotes 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 trap state to be discussed further below. That is,

\[
< S, A, \Delta, q > \xrightarrow{a} < S, A, \Delta, q' > \text{ iff } (q, a, q') \in \Delta \text{ and } q' \neq \pi.
\]

Since there is an one-to-one mapping between a process \( P \) and its LTS, the term process and LTS are used interchangeably. The above statement may also be rewritten as:

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

The state \( \pi \) is introduced for trapping a state at which specified safety properties are violated. A process that is trapped at the \( \pi \) state can participate into no further activities. Its behaviour is represented by an LTS \( < \{ \pi \}, \Sigma, \varnothing, \pi > \), where \( \Sigma \) represents the universal set of actions. For convenience, we use \( \Pi \) to denote a trapped process. Thus a process \( P = < S, A, \Delta, q > \) transits into \( \Pi \) if it executes a transition \( (q, a, \pi) \) in \( \Delta \). For instance, Figure 3 represents a Writer that transits into \( \Pi \) after executing update at state 0. Hence, Writer \( \xrightarrow{update} \Pi \).

![Figure 3: A Writer with a Trap State](image_url)

Behaviour of a process can be observed by means of its execution traces. An execution trace of a process is a sequence of actions that it can perform. For example, the sequence \( \langle \text{lock, update, unlock} \rangle \) is a trace of Writer in Figure 2. The set of possible traces of a process \( P \) is denoted by \( \text{tr}(P) \), whose formal definition can be found in the work of Hoare [18]. A trace is said to be trapping if its execution leads a process to \( \Pi \); otherwise it is nontrapping. A process is said to be trappable if it contains some trapping traces; otherwise it is nontrappable.

Observability of actions in a process can be controlled 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 trapping traces if and only if \( P \uparrow L \) has. Rules (1) and (2) in Figure 4 give the transitional semantics of the restriction operator. Processes in a distributed program may also be composed by a composition operator \( \| \) similar to that used in
CSP [18]. \( P \| Q \) 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 \( P \| Q \) is given by the union of their individual alphabets (i.e. \( \alpha_P \cap \alpha_Q \)).

1a. \( \frac{a \rightarrow P'}{P \rightarrow L} \) (\( a \in L, P' \neq \Pi \) )

1b. \( \frac{a \rightarrow \Pi}{P \rightarrow L} \) (\( a \in L \) )

2a. \( \frac{a \rightarrow P'}{P \rightarrow L \rightarrow \tau} \) (\( a \in L, P' \neq \Pi \) )

2b. \( \frac{a \rightarrow \Pi}{P \rightarrow L \rightarrow \tau} \) (\( a \in L \) )

3a. \( \frac{a \rightarrow P'}{P \| Q \rightarrow a \rightarrow P' \| Q} \) (\( a \notin \alpha_Q, P' \neq \Pi \) )

3b. \( \frac{a \rightarrow \Pi}{P \| Q \rightarrow a \rightarrow \Pi} \) (\( a \notin \alpha_Q \) )

4a. \( \frac{Q \rightarrow a \rightarrow Q'}{P \| Q \rightarrow a \rightarrow P \| Q'} \) (\( a \notin \alpha_P, Q' \neq \Pi \) )

4b. \( \frac{Q \rightarrow a \rightarrow \Pi}{P \| Q \rightarrow a \rightarrow \Pi} \) (\( a \notin \alpha_P \) )

5a. \( \frac{a \rightarrow P' \| Q \rightarrow a \rightarrow P' \| Q'}{P \| Q \rightarrow a \rightarrow P' \| Q'} \) (\( a \in \alpha_P \cap \alpha_Q, P' \neq \Pi, Q' \neq \Pi \) )

5b. \( \frac{a \rightarrow P' \| Q \rightarrow a \rightarrow \Pi}{P \| Q \rightarrow a \rightarrow \Pi} \) (\( a \in \alpha_P \cap \alpha_Q, P' = \Pi \) or \( Q' = \Pi \) )

**Figure 4: Rules of the Restriction and Composition Operators**

Rules (3), (4) and (5) in Figure 4 give the transitional semantics of the composition operator. The operator is both commutative and associative. Figure 5 shows the LTS of \( A \| B \) composed from processes \( A \) and \( B \) by applying the rules. The rules also specify that a composite process is trapped at state \( \pi \) if any of its constituent processes is trapped.

**Figure 5: Processes \( A, B \) and their Composite Process \( A \| B \)**

### 2.2 Behavioural Equivalences

The notion of *observation* on processes is postulated to describe the process behaviour conceived by an observer making experiments about processes. *Behaviour equivalence* is a concept to identify a pair of processes which cannot be distinguished by such observations.
Strong equivalence, denoted as \(\sim\), is used to relate two processes whose behaviours are indistinguishable to an observer by tests of interest to the theory of concurrency. Weak equivalence, denoted as \(\approx\), is used to relate two processes whose behaviours are indistinguishable to an observer when given their observable behaviours. Comprehensive treatment of strong and weak equivalences can be found in the work of Milner [26]. Let \(\wp\) and \(\Sigma\) be the universal set of processes and that of actions including \(\tau\), respectively. A strong equivalence \(\sim\) is the union of all relations \(R \subseteq \wp \times \wp\) satisfying that \((P, Q) \in R\) implies:

(i) \(\alpha P = \alpha Q\);
(ii) for all \(a \in \Sigma\):
   (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 \(P \xrightarrow{\tau} P'\). A weak equivalence \(\approx\) is the union of all relations \(R \subseteq \wp \times \wp\) satisfying that \((P, Q) \in R\) implies:

(i) \(\alpha P = \alpha Q\);
(ii) for all \(a \in \Sigma\):
   (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 \((P', Q') \in R\).

2.3 A Gas Station Example

As an illustration for our discussion, we present a gas station example (Figure 6) originally proposed by Helmbold and Luckham [17]. The example models an automated gas station...
with an operator, a pump, two customers and a queue holding customers’ requests\(^1\). The
design architecture for the gas station system in Figure 7 describes both the structure and the
interaction between the subsystems and their components. It is specified in an \textit{Architecture}
\textit{Description Language} (ADL) such as Darwin\(^2\) [22], which has a graphical representation as
shown in the figure. Interaction between computational components is achieved by
communication on bindings between matching pairs of service requirements (white dots) and
provisions (black dots). Subsystems are introduced to encapsulate internal structure and
interaction. The approach allows for a modular and comprehensible system design.
Decomposition of subsystems terminates at the primitive components which are represented
as shaded rectangles. Behaviours of primitive processes are given as LTSs to be realized in
some programming languages, such as C++.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{image}
\caption{The Design Architecture of the Gas Station System}
\end{figure}

\(^1\) The example can be extended to accommodate more operators, customers and pumps.

\(^2\) Darwin supports the definition of parameterised component types and generic architectures. For analysis and
program construction, a generic architecture must be instantiated, evaluating the parameters, elaborating the
component definitions and performing interface bindings to generate a particular architectural instance. This can
then be analysed or implemented in Regis [23], a distributed programming environment which supports Darwin
design specifications. For simplicity, we neglect many of the naming and elaboration issues.
The design architecture of the gas station postulates a hierarchy of subsystems as shown in Figure 8. In the hierarchy, subsystems are composite processes (represented as boxes with rounded corners) formed by composition of simpler subsystems or primitive processes (represented as boxes with sharp corners). For instance, it shows that the composite process Counter can be built from primitive processes Operator and Queue, and Station, from Counter and Pump. The primitive processes are leaves of the subsystem hierarchy.

**Figure 8: Subsystem hierarchy of the Gas Station System**

Figure 9 gives the LTSs presenting the behaviour of the primitive components operator and the request queue. The operator may initially choose to accept money prepaid by customers (prepay) or accept the amount to be charged from the pump (charge). 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 \( change = prepay - charge \) to the customer and activates the pump again if there are other customers waiting for the pump.

**Figure 9: Behaviour of the Operator and the Queue holding customers' requests.**
Figure 10 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 (\textit{finish}) and wait for the change from the operator. Upon receiving the “finish” request, the pump informs the operator of the charge information.

![Figure 10: Behaviour of the Pump and two customers Cust1 and Cust2](image)

As mentioned, hiding internal interaction helps make the design and analysis more comprehensible. Let us assume in the following discussion that the software developers are interested in the system behaviour with respect to actions \textit{prepay1} and \textit{prepay2}. In other words, only these two actions are observable in the global LTS of \textit{GasSystem}.

\[
\text{GasSystem} = (\text{Station} \parallel \text{Clients}) \uparrow \{ \text{prepay1}, \text{prepay2} \}
\]

3. Enhanced Compositional Reachability Analysis

3.1 Limitations of Compositional Reachability Analysis

Compositional derivation of the overall system behaviour has been shown to be a very promising approach for reachability analysis [30, 31, 37]. In such compositional reachability analysis (CRA) techniques, the model of the target system is given as an LTS which describes the overall system behaviour. Given a subsystem 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 \textit{GasStation} in Figure 8 can be composed in four steps. First, compose the LTS of \textit{Counter} from \textit{Operator} and \textit{Queue}. Second, use that LTS and \textit{Pump} to generate the LTS of \textit{Station}. Third, construct the LTS of \textit{Clients} from \textit{Cust1} and \textit{Cust2}. Fourth, compose the LTS of \textit{GasStation} from that of \textit{Station} and that of \textit{Clients}. This mechanism of "intermediate simplification during composition" is attractive for the analysis of modular systems. Figure 11 shows the global LTS thus obtained. It shows that the gas system repeatedly accepts money from \textit{Cust1} and \textit{Cust2} by means of actions \textit{prepay1} and \textit{prepay2}. There is no particular ordering relation between the occurrences of \textit{prepay1} and \textit{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 that involve other actions cannot be examined from the global LTS of the GasStation in Figure 11. Examples of these other properties are that the Operator must give the right change to the right customer (Figure 12(a)) and the Pump must complete the service to a customer before serving the other (Figure 12(b)). The former property refers to the expected behaviour of Operator and the latter to that of Pump. 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 mitigate against the 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

A safety property prescribes the set of legitimate traces. This can be represented by a deterministic property automaton $T = (A, S, \Delta, q)$, which is free of state $\pi$ and internal action $\tau$ (see Figure 12). A property is said to be violated, within the context of a distributed program, when some processes can perform a trace not acceptable to the property automaton. The detection mechanism is required to make the violation, if any, globally observable. To achieve this, we first derive an image automaton based on a given property automaton. The image automaton traps possible violation at the $\pi$ state. This image automaton is then composed with the processes to which the automaton applies. As the $\pi$ state is preserved by both restriction and composition operators, possible violation is thus made globally observable. Violation of safety properties can therefore be checked by examining the existence of the $\pi$ state 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:

1. initialise $\Delta'$ to $\Delta$

2. $(\forall a \in A$ and $s \in S$ where $(\not \exists s' \in S$ such that $(s, a, s') \in \Delta))$ add $(s, a, \pi)$ to $\Delta'$

![Figure 12: Two Property Automata Specified by Users](image1)

![Figure 13: Image Property Automata of (a) Right_Change and (b) Right_Service](image2)

For example, Figure 13 gives the corresponding image automata for Right_Change and Right_Service of Figure 12. The image automaton so constructed satisfies two conditions:

(i) $T$ and $T'$ have the same set of nontrapping traces (i.e. $\text{tr}(T)$); and

(ii) for any process $P, P || T'$ does not contain trapping traces iff $\text{tr}(P \uparrow \alpha T') \subseteq \text{tr}(T')$.

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.

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

Since $T$ and $T'$ share the same initial state $q$, all nontrapping 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 nontrapping traces, which is equal to $\text{tr}(T')$. \square

Proof of condition (ii)

Case (1): if part: Assume $\text{tr}(P \uparrow \alpha T') \subseteq \text{tr}(T), \text{tr}(P || T') \uparrow \alpha T' = \{t | t \in \text{tr}(P \uparrow \alpha T') \cap \text{tr}(T')\}$ which in turn equals $\{t | t \in \text{tr}(P \uparrow \alpha T') \cap \text{tr}(T')\} \cup \{t | t \in \text{tr}(P \uparrow \alpha T') \cap (\text{tr}(T') - \text{tr}(T))\}$. The image process $T'$ is so
constructed that only those traces in $tr(T')-tr(T)$ can lead $T'$ in a trap state. Hence, $\{ t \mid t \in tr(P^\uparrow \alpha T') \cap (tr(T')-tr(T)) \}$ represent the set of nontrapping and trapping traces respectively in $(P \parallel T')^\uparrow \alpha T$. As $tr(P^\uparrow \alpha T') \subseteq tr(T')$, $tr(P^\uparrow \alpha T') \cap (tr(T')-tr(T))$ is an empty set. As a result, there are no trapping traces in $(P \parallel T')^\uparrow \alpha T'$, $(P \parallel T')^\uparrow \alpha T'$ is nontrappable. Thus, $P \parallel T'$ is also nontrappable.

Case (2): only-if part: Assume $P \parallel T'$ is nontrappable. Let us suppose $tr(P^\uparrow \alpha T') \not\subseteq tr(T')$. This implies that $P^\uparrow \alpha T'$ can perform a trace $t$ that does not belong to $tr(T)$. We note that any prefix of $t$ can be performed by $P^\uparrow \alpha T'$. Let $s$ be a prefix of $t$ such that $T$ can perform all prefixes of $s$ but not the $s$ itself. By the method we construct the image process, $s$ can also be performed by $T$. As a result, $s$ is a trapping trace in $T$. Thus $s$ is a trapping trace in $(P \parallel T')^\uparrow \alpha T'$, which is equal to $(P \parallel T')^\uparrow \alpha T'$. By the semantics of restriction operator, the existence of a trapping trace in $(P \parallel T')^\uparrow \alpha T'$ implies the existence of a trapping trace in $P \parallel T$. This contradicts to the assumption that $P \parallel T'$ is nontrappable. Thus, the supposition cannot hold.

Condition (ii) enables us to detect violation of safety properties in a system by checking the existence of trapping traces in the composite process formed by the system and the image property automata. If trapping traces exist in the composite process, some safety properties are violated.

**Figure 14: The subsystem hierarchy for checking safety properties Right_Change' and Right_Service' which are associated with the Operator and Pump respectively**

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 14 shows the modified subsystem hierarchy to include the image property automata of Figure 13. As shown in the modified hierarchy, Right_Change’ and Right_Service’ are combined respectively with the processes Operator and Pump to which these properties refer. Figure 15 gives the global LTS derived by the CRA based on the hierarchy in Figure 14. Since the global LTS contains trapping traces, the Gas Station system does not satisfy both safety properties represented by the property automata Right_Change and Right_Service.
4. Debugging

4.1 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 needs to be further enhanced with a mechanism to keep track of the relation between those transitions leading to the trap state $\pi$ in the global LTS and those in the image property automata.

Let $\mu_P \subseteq ((\mathcal{S} - \{\pi\}) \times \Sigma) \rightarrow 2((\mathcal{S} - \{\pi\}) \times \Sigma)$ be a mapping associated with a process $P$, where $\mathcal{S}$ and $\Sigma$ denote the universal sets of states and actions, respectively. The value of $\mu_P(s, a, \pi)$ indicates the set of trapping transitions in the image interface processes that contribute to the transition $(s, a, \pi)$ in $P$.

The mapping $\mu_P$ is so evaluated that $\mu_P(s, a, \pi)$ equals the empty set $\emptyset$ if $(s, a, \pi)$ is a transition not reachable from the initial state in $P$; otherwise:

(i) when $P$ is an image property automaton,

$$\mu_P(s, a, \pi) = \{(s, a, \pi)_P\}$$

(ii) when $P = Q||R$,

$$\mu_P(s_Q, s_R, a, \pi) = \mu_Q(s_Q, a, \pi) \cup \mu_R(s_R, a, \pi)$$

(iii) when $P = Q^\uparrow L$,

$$\mu_P(s, a, \pi) = \mu_Q(s, a, \pi) \text{ for } a \in L$$

$$\mu_P(s, \tau, \pi) = \mu_Q(s, \tau, \pi) \cup \{v \mid v \in \mu_Q(s, a, \pi) \text{ for some } a \in \alpha Q-L\}$$

(iv) when $P$ is a simplification of $Q$ so that $P \approx Q$,

$$\mu_P(s, a, \pi) = \{v \mid v \in \mu_Q(s', a, \pi) \text{ for some } s' \text{ in } Q \text{ which identifies a process weakly equivalent to that identified by } s\}.$$  

The subscript $P$ in a transition $(s, a, \pi)_P$ indicates its owner process. For instance $Right\_change'$ is the process which owns the transition $(0, change1, \pi)_{Right\_Change'}$. Since the set union operation can be computed using bitwise arithmetic in imperative programming languages, its computational complexity is linear in time and space. It does not increase the computational complexity of the error detection mechanism.
Figure 15: The global LTS derived by the CRA based on the hierarchy in Figure 14

The global LTS given by the CRA based on the hierarchy is given in Figure 15. Using the above evaluation rules of $\mu_{P}$, $\mu_{GasStation}$ takes a value of $\{(1, change_2, \pi)_{Right\_Change}$, $(2, change_1, \pi)_{Right\_Change}\}$. This suggests that safety violation occurs at transitions $(1, change_2, \pi)$ and $(2, change_1, \pi)$ in image property automata $Right\_Change$ (Figure 13a). The former represents the situation where $charge_1$ can be followed by $change_2$ and the latter represents the situation where $charge_2$ can be followed by $change_1$. In either situation, customers receive the wrong change. On the other hand, the safety property specified by property automaton $Right\_Service$ (Figure 12b) can be ensured by the Gas Station.

4.2 Construction of Debugging Traces

A common error reporting technique in behavioural analysis is to present a debugging trace that leads the system being analyzed to exhibit undesirable properties. Users may then follow the given trace to get a deeper understanding of reported errors. While analysis of large systems is better done incrementally with details suppressed at intermediate stages, information provided by debugging traces should be as detailed as possible. A debugging trace should show explicitly how each primitive components of a system reaches a state where a desirable property is violated. Thus, a debugging trace of the $GasStation$ system may include unobservable actions performed by subsystems $Station$, $Clients$, $Pump$ and $Counter$. An example leading to the violation of $Right\_Change$ is $(prepay_1, pump\_avail, activate, start_1, finish_1, charge_2, change_1)$. Unobservable actions in a trace can be recovered using a reconstruction algorithm proposed by Yeh and Young [38]. The algorithm successively reconstructs unobservable actions in a trace according to the subsystem hierarchy employed for compositional analysis. Reconstruction terminates at points where it hits a delimited state, which is the trapping state $\pi$ in the current approach.
4.3 Regression Checking by Re-structuring

Debugging facilities of an analysis tool should ideally go beyond the generation of debugging traces. Facilities should be provided to reduce the effort in fixing errors. To fix a reported error, designers may choose to modify the specification of some components based on their intuition and preferences. For instance, the designer of the GasStation system may choose the Pump component and modify it trying to fix the violation of Right_Change. At the same time, the modified system should continue to satisfy the other desirable property Right_Service. To ensure this, the system needs to be checked again each time it has been modified. We refer this to as regression checking since error debugging is likely to be a trial-and-error exercise. For instance, the Pump specification may need be modified many times before the error is actually fixed. The effort of regression checking could contribute to a significant part of the software development cost. This effort can, to some extent, be relieved by the use of compositional analysis where only those subsystems affected by a change need be checked again. For example in Figure 16, only subsystems N0, N1 and N5 need be reconstructed in the subsystem hierarchy when the primitive components colored in black are modified. However the effort of regression checking may still be expensive for large systems where many subsystems are affected.

A more effective approach is to avoid regression checking of subsystems. To achieve this, we need to re-structure the subsystem hierarchy and construct a simplified, composite LTS for all components in the system other than the subsystem being modified. The re-structuring algorithm is presented in Figure 17.
procedure re-structure( (V, E), V_c )
input: (V, E): a tree representing the original subsystem hierarchy
V_c: a set of primitive components to be modified
output: a tree (V', E') representing the re-structured subsystem hierarchy
begin
1. V' := V ∪ {v_o'}, where v_o' ∈ V; (* v_o' is the new root of tree (V', E') *)
2. E' := E ∪ {(v_o', v_o)}; (* v_o is the old root of tree (V, E) *)
3. for each v ∈ V_c do
4. E' := E' ∪ {(v_o', v)};
5. endfor
6. V_s := sibling_{(V, E)}(V_c) - V_c;
7. for each v ∈ V_s do
8. E' := E' ∪ {(v_o, v)}; (* attach siblings to the old root *)
9. endfor
10. V_p := parent_{(V, E)}(V_c); (* set of immediate parents of vertices in V_c *)
11. for each v ∈ V_p do
12. remove v from (V', E');
13. endfor
end

Figure 17: Re-structuring Algorithm

The re-structure algorithm takes in a tree (V, E) that records the original subsystem hierarchy, where V and E represent the vertices and edges, respectively. Upon termination, it returns a re-structured hierarchy recorded as (V', E'). Statements 1 and 2 insert a new root v_o' in (V', E') and link it to the old root v_o of (V, E). The new root v_o' is then made to be a node that links directly to v_o and those modified primitive components in V_c. Afterwards, the old root v_o is made to be a subsystem that holds all components in the system other than the ones in V_c. The function sibling_{(V, E)}(V_c) returns a set of primitive components in (V, E) that share the same immediate parents as those in V_c. As these sibling components are placed under the same immediate subsystems with those being modified, they are likely to have close interaction with each other. As such, sibling components are better placed as near as possible to the modified ones. This is achieved by putting these sibling components directly under the old root v_o'. The function parent_{(V, E)}(V_c) returns a set of immediate parents of vertices in V_c. With all immediate successors having been relinked, these parent nodes can be removed from the tree. Note that the algorithm may also be applied to cases where primitive components are scattered in the subsystem hierarchy and are embedded in more than one immediate subsystem.
Using the re-structuring algorithm, the hierarchy in Figure 16 is transformed to that in Figure 18 which is used for regression checking. In the first round of regression checking, the number of subsystems that need to be reconstructed in the re-structured hierarchy is no more than that required in the original hierarchy. The computational effort of regression checking for both structures is thus comparable even if errors can be fixed at the first attempt. The effort of regression checking is significantly reduced in subsequent attempts because only the LTS of the new root \( N0' \) needs be re-constructed regardless of the tree size and structure. The approach of regression checking using re-structuring is particularly useful if the changes affect many subsystems.

![Figure 18: A Re-structured Subsystem Hierarchy](image)

Clearly, this re-structuring approach does not necessarily work if subsystems are constructed using conventional CRA, without the inclusion of trapping states. This is because the composite LTS constructed may only check properties involving actions at the interface of the modified subsystem [10]. To illustrate the point, let us consider the gas control system as an example and suppose that \( \text{Pump} \) is the primitive component to be modified. The property \( \text{Right}_\text{Change} \) which involves actions not belonging to the interface of \( \text{Pump} \) cannot be checked using a re-structured hierarchy. As a result, the enhanced CRA technique needs to be employed for regression checking. The inclusion of the \( \pi \) state allows the satisfiability conditions of subsystem properties to be propagated up a subsystem hierarchy even though these properties may involve hidden actions. These properties are satisfied if and only if the LTS constructed for the root in the subsystem hierarchy is nontrappable. In the re-structuring algorithm, property automata are treated as primitive components that are not to be modified.
4.4 Regression Checking of the Gas Control System

Suppose \textit{Pump} is the component selected to be modified to eradicate the reported error. Using the re-structuring algorithm, one can derive a subsystem hierarchy as shown in Figure 19 for regression checking from that given in Figure 14. The composite LTS \textit{Counter&Clients} is constructed in the first round of regression checking. The LTS is simplified by hiding all actions not involved in the interaction with \textit{Pump}. After simplification, it consists of 48 states and 103 transitions. Subsequent rounds of regression checking can be conducted in one step by composing the LTS of \textit{Counter&Clients} and that of the modified \textit{Pump}. If the LTS derived from the composition of \textit{Counter&Clients} and the modified \textit{Pump} is nontrappable, one can conclude the satisfaction of both properties \textit{Right_Change} and \textit{Right_Service}. This is the case if the specification of \textit{Pump} is modified to that shown in Figure 20. In fact, the \textit{GasStation'} formed by composing the \textit{Counter&Clients} and the \textit{Pump} was found to behave like the \textit{Pump} itself. Figure 21 gives the global LTS based on the subsystem hierarchy in Figure 14 after modification.

![Figure 19: A Re-structured Subsystem Hierarchy of the Gas Control Station for Regression Checking](image1)

![Figure 20: The Modified Pump](image2)
4.5 Correctness of the Global LTS

In previous work [4], 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 22). 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 trapping traces if and only if $tr(Z \uparrow \alpha Ifc) \subseteq tr(Ifc)$. As a result, the absence of reachable trap 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 trapping traces\footnote{Strong equivalence is a subrelation of weak equivalence (c.f. Milner [20]).}. For instance, Figure 21 gives the global LTS constructed with the inclusion of $Right\_Change'$ and $Right\_Service'$. Since the LTS does not contain any trapping traces, it represents the overall behaviour of $GasStation$ that satisfies both properties.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{global_lts.png}
\caption{The Global LTS of GasStation}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{interface_theorem.png}
\caption{Interface Theorem}
\end{figure}

Suppose $Z$ and $Ifc$ are two processes; and $\sim$ denotes the strong equivalence relation.

\[
Z \sim (Z \parallel Ifc)
\]

if

\begin{enumerate}
  \item $\alpha Ifc \subseteq \alpha Z$;
  \item $tr(Z \uparrow \alpha Ifc) \subseteq tr(Ifc)$;
  \item $Ifc$ is a deterministic process free of internal action $\tau$.
\end{enumerate}
4.6 Early Detection of Infeasible Fixes

There is an additional benefit in re-structuring. Designers may confirm at an early stage if the set of selected components can be modified to fix the reported errors. For instance, designers may readily confirm that the modification of the selected components alone cannot help fix the errors when the LTS made up by other components in the system and the property automata contains a trapping trace involving only $\tau$ actions. Thus, no matter how the designers modify the selected components, the trapping trace remains in the global LTS.

5. Performance

A prototype based on the concepts discussed has been implemented on a Sun Sparc/20 Server [13]. Figure 23 and Figure 24 give the computational time in seconds required to compute the global LTS of a faulty and correct gas example, respectively. The first column indicates the number of customers in the gas station. The second and third columns give the computational time required to evaluate the global LTS based on the compositional hierarchy in Figure 8 and Figure 14, respectively. We found that an incorrect specification could significantly increase the computational costs in validating safety properties. This suggests that the analysis technique should be further improved before it can handle realistic systems. A possible way to reduce the computational costs is to combine the technique with various state space reduction mechanisms, such as those proposed by Godefroid [14], and the previous work of the authors [4, 9].

<table>
<thead>
<tr>
<th>#Customers</th>
<th>Before the inclusion of the two property automata in Figure 13</th>
<th>After the inclusion of the two property automata in Figure 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.6</td>
<td>0.7</td>
</tr>
<tr>
<td>3</td>
<td>1.5</td>
<td>2.1</td>
</tr>
<tr>
<td>4</td>
<td>7.8</td>
<td>19.0</td>
</tr>
<tr>
<td>5</td>
<td>80.1</td>
<td>290.3</td>
</tr>
<tr>
<td>6</td>
<td>1,003.5</td>
<td>5,068.3</td>
</tr>
</tbody>
</table>

Figure 23: Computational Time (in seconds) Required to Evaluate the Global LTS of the Faulty Gas System
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 trap 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 occur. The mechanism may be further optimised by augmenting the CRA technique with the concept of context constraints [4, 7, 9] and partial ordering [14]. These constraints capture behavioural restriction imposed on subsystems by their neighbouring processes.

To further explore the potential of the technique, we are hoping to apply it to more complex examples. Further work is needed to provide guidance as to which actions to hide and at which point in the compositional hierarchy to compose the properties to be checked. This is both a logical decision as to which is the most sensible, and an efficiency decision as to which aids the minimisation automation. Supporting analysis tools [13] have been implemented, and we are proposing to incorporate these into an environment for the design and construction of distributed programs, the System Architect's Assistant [21] which utilises the Darwin language [23] for the specification of the software architectures. We also

<table>
<thead>
<tr>
<th>#Customers</th>
<th>Before the inclusion of the two property automata in Figure 13</th>
<th>After the inclusion of the two property automata in Figure 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.8</td>
<td>0.7</td>
</tr>
<tr>
<td>3</td>
<td>1.9</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>11.8</td>
<td>11.9</td>
</tr>
<tr>
<td>5</td>
<td>123.2</td>
<td>129.6</td>
</tr>
<tr>
<td>6</td>
<td>1,540.9</td>
<td>1,611.4</td>
</tr>
</tbody>
</table>

Figure 24: Computational Time (in seconds) Required to Evaluate the Global LTS of the Correct Gas System

6. Conclusion and Future Work
considering developing a framework to integrate this enhanced CRA technique with a data flow analysis technique [5, 6]. Finally we are investigating an approach for the specification and checking of liveness conditions which is to and can be used together with our enhanced CRA approach for safety properties described in this paper.

7. Acknowledgment

We are indebted to Dimitra Giannakopoulou for interesting discussions and invaluable suggestions and for her contribution to the analysis tools. We also wish to express our thanks for financial support for the work by the following grants: RGC Grant HKUST 662/95E and EPSRC Grant GR/J 87022 (TRACTA Project).

8. References


