Compositional Reachability Analysis
of Finite-State Distributed Systems
with User-Specified Constraints

Shing-chi Cheung* and Jeff Kramer†

Technical Report HKUST-CS95-3
January 1995

*Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
(email: scc@cs.ust.hk)

†Department of Computing
The Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, United Kingdom
(email: jk@doc.ic.ac.uk)
Compositional Reachability Analysis of Finite-State Distributed Systems with User-Specified Constraints

Shing-chi Cheung                Jeff Kramer

Department of Computer Science  Department of Computing
Hong Kong University of Science and Imperial College of Science, Technology and Medicine, London SW7 2BZ, UK
Technology, Clear Water Bay, Hong Kong

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. This staged composition may however exacerbate the state explosion problem. A promising way to alleviate this problem is to include, in the analysis of each subsystem, the behaviour constraints imposed by its environment. Behaviour constraints can be derived algorithmically but may be too weak to adequately constrain the compositional analysis. In this paper, we describe a technique which enables software developers to also specify behaviour constraints intuitively. These constraints can be combined with those derived algorithmically. As constraints specified by software developers may be incorrect, the technique incorporates an effective mechanism to detect incorrect constraints and give an indication of how to correct them. Incorporation of the mechanism does not increase the complexity of analysis. The mechanism is supported by augmenting the state machine model with a special state $\pi$. The state is used to capture automatically potential errors in the constraints specified by software developers. The mechanism can also be utilised to detect violation of design assumptions. A gas station system is used as a case study in our discussion.
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.

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 [4, 8, 15]. In the latter category, reduction is achieved by intermediate simplification of subsystems [10, 12-14, 17]. Techniques in this category are known as *compositional reachability analysis*. However, these analysis techniques do not consider environment constraints in the construction of LTSs for subsystems. These environment constraints, known as *context constraints*, are behaviour restrictions imposed on each process by its neighbours via the need for communication synchronisation. As a result, these LTSs may be excessively large
and contain many spurious traces. As such, the state explosion problem could be exacerbated.

Contextual reachability analysis is a promising technique to alleviate the problem by including context constraints in compositional minimisation [1, 5, 16]. These constraints can be derived algorithmically [1] but may be too weak to adequately constrain the compositional analysis. Users may therefore wish to provide their own constraints based on their knowledge and intuition about the target system. This approach was studied by Graf [5] and Yeh [16]. The technique of Graf allows users to specify context constraints as finite-state interface processes which introduce undefinedness predicates wherever the processes inhibit some transitions. Since users may possess an imprecise knowledge of the system, interface processes given by users may be incorrect. To cater for this, the technique flags an error at the end of the analysis if it has included some erroneous interface processes. The technique of Graf however requires an additional step of backward projection which may introduce spurious traces to subsystems and thus reduce the effectiveness of contextual reachability analysis. The technique of Yeh allows users to insert start and stop primitives to impose context constraints. However, the use of these primitives makes the parallel composition operator non-associative.

Both techniques of Graf and Yeh have not included a mechanism to trace back the errors and transitions that are missing in the erroneous interface processes. This poses a problem if users wish to correct the interface processes and conduct the analysis technique again. To tackle this problem, we have refined the technique of Graf and introduced an error detection mechanism that is capable of identifying erroneous interface processes and locating the source of errors. The mechanism is supported by augmenting the state machine model with a special undefined state $\pi$. The state is used to capture potential errors in the constraints specified by users. The mechanism is effective in the sense that it does not increase the complexity of the contextual reachability analysis and does not require the backward projection step as in the technique of Graf. The compositional operator in our technique is both commutative and associative. In the paper, we also show how context constraints provided by users can be combined with those derived algorithmically for contextual reachability analysis, and how to utilise the mechanism to detect violation of design assumptions.

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 include context constraints in compositional reachability analysis. An account of a mechanism to detect erroneous context constraints is also given. Section 4 gives an illustrative example of the use of the technique to check violation of design assumptions. This is followed by conclusions in
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 1 represents an LTS of a process 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 prepay1 and back to state 0 from 1 by an action charge1. Alternatively, the process can go back to state 0 from 0 by executing charge2.

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 \{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 $\langle S, A, \Delta, q \rangle$ 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 $P = \langle S, A, \Delta, q \rangle$ transits into another LTS of $P' = \langle S, A, \Delta, q' \rangle$ 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 to be discussed
shortly. That is,  
\[ < S, A, \rightarrow, q > \xrightarrow{a} < S, A, \rightarrow, q' > \iff (q, a, q') \in \Delta \text{ and } q' \neq \pi. \]

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

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

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 \( \langle \text{prepay1, charge1, charge2} \rangle \) is a trace of \( \text{Proc} \) in figure 1. 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 [7]. 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 2(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,  
\[ < S, A, \rightarrow, q > \xrightarrow{a} < \{\pi\}, A, \emptyset, \pi > \iff (q, a, \pi) \in \Delta. \]

Alternatively, we may write:

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

For instance, figure 2(b) represents a process \( \text{Proc} \) that transits into \( \Pi \) after executing \( \text{charge1} \). Hence, \( P\xrightarrow{\text{charge1}}\Pi \).

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 of which turns a process into \( \Pi \). For instance, \( \langle \text{charge2, charge1} \rangle \) 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 handle the situations where
specified system behaviour is not consistent with that perceived by users.

In the LTS computational model, actions of a process can be restricted by a restriction operator ↑. $P \uparrow L$ 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.

![Figure 3: Processes A, B and their Composite Process $A \|$ $B$](image)

Processes in a distributed system can also be composed by a composition operator $\|$ similar to that used in CSP [7]. $P \|$ $Q$ 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. $\alpha P \cup \alpha Q$).

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

The restriction operator $\uparrow$ can be distributed into the composition operator $\|$ 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 $L, M$:
   \[(\alpha P - L) \cap (\alpha Q - M) = \emptyset \text{ implies } P \uparrow L \parallel Q \uparrow M = (P \parallel Q) \uparrow (L \cup M)\]

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) $P = \prod$ if and only if $Q = \prod$.

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

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

(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 $(P', Q') \in R$.

The strong and weak semantic equivalences coincide with the strong and weak equivalence (cf. Milner [11]) 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) $P \approx Q$ 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 [6]. The system models an automated gas station with an operator, a pump, two customers and a queue holding customers' requests. Figure 4 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 5 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 *Operator, Queue, Pump, Cust1* and *Cust2* using a compositional hierarchy as shown in figure 6. 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:

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

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. Contextual Compositional Reachability Analysis

#### 3.1 Inclusion of Context Constraints

Recent literature has reported promising results by employing a compositional approach to generate the model of a system using reachability analysis [12, 13, 17]. The model is given as

---

\(^1\) If necessary, the system can be extended to accommodate more customers and pumps.
Figure 7(a): Component Processes of Counter    Figure 7(b): Example Interface Processes

Although this mechanism of "intermediate simplification during composition" is attractive for the analysis of modular systems, it has a drawback in that the state explosion problem could be exacerbated. The LTSs of subsystems are constructed without considering their interaction with neighbouring processes in the system [5]. As a result, LTSs of subsystems may contain many behavioural traces that can never be executed in the system as a whole. Researchers have thus proposed techniques to tackle this problem by including context constraints in the construction of LTSs for subsystems [1, 5, 16]. Context constraints are behaviour restrictions imposed on the subsystem by its neighbouring processes. This technique is known as contextual reachability analysis.

For example, suppose we wish to construct an LTS for subsystem Counter. Originally, the subsystem is composed of two primitive processes Operator and Queue. However, the LTS constructed by simply composing the LTSs of Operator and Queue contains spurious behavioural traces that are forbidden by processes Pump, Cust1 and Cust2. In the contextual reachability analysis, we imagine that Counter is composed of Operator, Queue and an interface process Ifc as shown in figure 7(a). The interface process Ifc is used to abstract context constraints imposed on Counter by Pump, Cust1 and Cust2. To ensure that the inclusion of Ifc does not affect the overall behaviour of GasStation, Ifc has to satisfy the interface theorem in figure 8 [1]. The theorem gives the sufficient conditions for a process Z to behave the same as
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 8: Interface Theorem

$Z \parallel Ifc$. The first condition ensures that $Ifc$ does not introduce any actions that cannot be performed by $Z$. The second condition ensures that inclusion of $Ifc$ does not eliminate any valid traces of $Z$. The third condition ensures that $Ifc$ does not introduce extra non-determinism or divergence to $Z$.

To apply the theorem, we assume a system $Z$ formed by the parallel composition of the primitive processes of the target system. For example in the case of $GasStation$, we assume a system $Z$ given by $(Operator \parallel Queue \parallel Pump \parallel Cust1 \parallel Cust2)$. By rule (6) in section 2, $Z \uparrow \{prepay1, prepay2\}$ behaves the same as $GasStation$. As a result, the behaviour of $GasStation$ can be given by the composed LTS formed by $(Z \parallel Ifc) \uparrow \{prepay1, prepay2\}$. Therefore, $Operator$, $Queue$ and $Ifc$ can be grouped together in the evaluation of the behaviour of $GasStation$. Candidate interface processes satisfying the interface theorem can be derived algorithmically [1]. Figure 7(b) shows three candidate interface processes derived algorithmically for the composition of the LTS of $Counter$. In fact, these interface processes can be composed together forming another interface process. For instance, $Ifc$ in figure 7(a) is a composite process of $Ifc0$, $Ifc1$ and $Ifc2$ in figure 7(b).

In many situations, users have further knowledge and intuitions of the system behaviour. Users may therefore be able to specify the behaviour constraints imposed by the environment on some subsystems. This information can be exploited during the contextual reachability analysis by means of interface processes. These user-supplied interface processes can be composed directly with those derived algorithmically for effective reduction of state spaces.

The main problem with this approach is that interface processes specified by users could be incorrect. These interface processes may violate the conditions in the interface theorem.
Inclusion of these incorrect processes in the analysis results in violation of system behaviour: the global LTS thus constructed does not faithfully represent the behaviour of the target system. Therefore a mechanism is incorporated to identify incorrect interface processes and provide an indication of how to correct them.

### 3.2 Error Detection Mechanism

Violation of the first and third conditions in the interface theorem can be readily identified. To check the violation of the second condition, the error detection mechanism refines an interface process $I = (A, S, \Delta, q)$ provided by users into another process $I' = (A, S \cup \{\pi\}, \Delta', q)$. The refined interface process $I'$ accepts all traces formed by actions in the alphabet of $I$ but those which cannot be exhibited by $I$ become undefined in $I'$. The refinement can be performed using the following procedure:

(i) initialise $\Delta'$ to $\Delta$

(ii) for all $a \in A$ and $s \in S$ where there does not exist $s' \in S$ such that $(s, a, s') \in \Delta$:

$$\text{add } (s, a, \pi) \text{ to } \Delta'$$

$I'$ is so refined that, when given a process $Z$ where $\alpha I' \subseteq \alpha Z$, all traces in $\text{tr}(Z) - \text{tr}(Z||I')$ are flagged as undefined in the composite process $Z||I'$. Hence, $Z||I'$ contains undefined traces if and only if $\text{tr}(Z)$ is not a subset of $\text{tr}(Z||I')$; i.e. $\text{tr}(Z\uparrow \alpha I) \notin \text{tr}(I')$. In order words, $Z||I'$ is free from undefined traces if and only if $\text{tr}(Z\uparrow \alpha I) \subseteq \text{tr}(I')$. This property of $I'$ in turn also holds for $Z||I'$ and other processes composed using $I'$. Figure 9 shows the original interface process $Iu$ supplied by users and the one $Iu'$ refined by the error detection mechanism for the contextual reachability analysis of $\text{GasStation}$. Note that $Iu$ and $Iu'$ have the same set of defined traces.

The refined interface process $Iu'$ is then used, instead of $Iu$, for the contextual reachability analysis of $\text{GasStation}$. Any valid traces of $\text{GasStation}$ that have been eliminated in the analysis become undefined in the global LTS. Since the undefinedness of traces is preserved by the restriction operator, an absence of undefined traces from the global LTS means an absence of undefined traces from $Iu'||Z$, where $Z$ equals $(\text{Operator} \parallel \text{Queue} \parallel \text{Pump} \parallel \text{Cust}1 \parallel \text{Cust}2)$.
Given that $\alpha Iu'$ is a subset of $\alpha Z$, this implies $tr(Z \uparrow \alpha Iu') \subseteq tr(Iu')$. As a result, $Iu'$ and $Z$ satisfy the interface theorem, and $(Z \parallel Iu')$ behaves equivalently to $Z$. The global LTS obtained in the contextual reachability analysis thus faithfully reflects the behaviour of $GasStation$.

In general, users may specify a number of interface processes at various stages of the contextual reachability analysis to express their knowledge of system behaviour related to different subsystems. When the global LTS has a reachable undefined state, some of its legal traces have been eliminated by the given interface processes. Error correction requires users to re-specify the interface processes. However in many situations users may not be able to identify exactly which transitions are missing. For the error detection mechanism to be useful, this kind of information should be provided automatically.

To provide this information, the error detection mechanism can be designed to keep track of the relation between those transitions leading to the undefined state $\pi$ in the global LTS and those missing in the interface processes specified by users.

For convenience, we use $[P \xrightarrow{a} \prod]$ to denote the set of transitions in the refined interface processes that contributed to the transition of $P \xrightarrow{a} \prod$. We call $[P \xrightarrow{a} \prod]$ the set of ancestor transitions of $P \xrightarrow{a} \prod$. For instance, let $Iu_0'$ be $Iu'$ and $Iu_0' \xrightarrow{\text{prepay1}} Iu_1'$ in figure 9(b). The values of $[Iu_0' \xrightarrow{\text{activate}} \prod]$ and $[Iu_1' \xrightarrow{\text{prepay1}} \prod]$ are $\{(0, \text{activate}, \pi)_{Iu'}\}$ and $\{(1, \text{prepay1}, \pi)_{Iu'}\}$ respectively. The subscript $Iu'$ identifies the LTS that owns the transition. In the contextual reachability analysis, processes are successively composed, restricted to some actions and then minimised based on the rules in section 2. In each of these analysis steps, the set of ancestor transitions are updated as follows:

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

- $[P \parallel Q \xrightarrow{a} \prod]$ to $[P \xrightarrow{a} \prod]$ if $Q \xrightarrow{\tau} \prod$
- $[P \parallel Q \xrightarrow{a} \prod]$ to $[Q \xrightarrow{a} \prod]$ if $P \xrightarrow{\tau} \prod$
- $[P \parallel Q \xrightarrow{a} \prod]$ to $[P \xrightarrow{a} \prod] \cup [Q \xrightarrow{a} \prod]$ if $P \approx Q$

(b) In the step of action restriction, update

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

(c) In the step of process minimisation, update

both $[P \xrightarrow{a} \prod]$ and $[Q \xrightarrow{a} \prod]$ to $[P \xrightarrow{a} \prod] \cup [Q \xrightarrow{a} \prod]$ if $P \approx Q$. 

14
3.3 An Example

Suppose \( Iu \) in figure 9(a) represents an interface process given by users to express their intuitive understanding of the behaviour constraints imposed on subsystem \( \text{Counter} \) by its neighbouring processes. The process is refined into \( Iu' \) in figure 9(b) to cater for any potentially missing transitions. This refined process can then be combined with those derived algorithmically in the construction (i.e. \( \text{Ifc0}, \text{Ifc1}, \text{Ifc2} \)) of a composite LTS for \( \text{Counter} \) as shown in figure 10(a). The LTS of \( \text{Counter} \) is therefore given by a parallel composition of \( \text{Operator}, \text{Queue}, Iu', \text{Ifc0}, \text{Ifc1} \) and \( \text{Ifc2} \). After simplification and hiding internal actions, the LTS of \( \text{Counter} \) is shown in figure 10(b). Let \( \text{Counter}_0 \) be \( \text{Counter} \) and \( \text{Counter}_0 \xrightarrow{\text{prepay2}} \text{Counter}_2 \). Using the above mechanism, the value of \([\text{Counter}_2 \xrightarrow{\text{activate}} \pi]\) is derived to be \{(0, activate, \pi)_{Iu'}\}. This indicates that the transition of \( \text{Counter}_2 \xrightarrow{\text{activate}} \pi \) in \( \text{Counter} \) is caused by the transition of \( Iu' \xrightarrow{\text{prepay1}} \pi \) in \( Iu' \).

Note that the transition of \( Iu' \xrightarrow{\text{prepay1}} \pi \) does not introduce any undefined traces in \( \text{Counter} \). That transition is made impossible by the behaviour of \( \text{Operator} \) who requires that any two consecutive executions of \( \text{prepay1} \) have to be separated by an execution of \( \text{activate} \).

Figure 11 gives the global LTS obtained in the contextual reachability analysis. Let \( \text{GasStation}_0 \) be \( \text{GasStation} \) and \( \text{GasStation}_0 \xrightarrow{\text{prepay2}} \text{GasStation}_1 \). Then, the value of \([\text{GasStation}_1 \xrightarrow{\tau} \pi]\) is derived to be \{(0, activate, \pi)_{Iu'}\}. The existence of the transition of \( \text{GasStation}_1 \xrightarrow{\tau} \pi \) in \( \text{GasStation} \) indicates that the LTS does not faithfully represent the behaviour of the system. This due to the omission of transitions in \([\text{GasStation}_1 \xrightarrow{\tau} \pi]\); i.e. an outgoing transition labelled with \( \text{activate} \) from state 0 in the interface process \( Iu \). The information helps users to track the error down to particular transitions. A correct specification
of $Iu$ is given in figure 12(a). The global LTS of $GasSystem$ derived using $Iu'$ in figure 12(b) is given in figure 13. Since the LTS is free of reachable undefined state, it correctly reflect the behaviour of $GasSystem$.

The error detection mechanism can be fully automated and it does not change the order of complexity of the contextual reachability analysis technique since the sizes of the subsystems increase by at most one to accommodate the undefined state $\pi$.

4. Exception Raising for Violation of Design Assumptions

In some situations, software developers may specify the behaviour of a subsystem based on certain assumptions of its environment. For instance, the LTS of Operator carries no information that Cust1 always receives the appropriate change. The developer of the process Operator in figure 4 has not specified explicitly that charge1 must be followed by change1 rather than change2. The assumption that Cust1 and Cust2 always receive the appropriate change has been made in the specification of Operator. Thus the correctness in the behaviour of Operator relies on the validity of this assumption. In this situation, it is useful to state the assumption and detect if there is any violation during the contextual reachability analysis. Design assumptions can be considered as an interface process specified by users. For example, figure 14(a) shows the above assumption made at the specification of Operator. The error detection mechanism can be used to check violation of assumptions. The design assumption is refined into another LTS as discussed in section 3.2 by adding transitions leading to the undefined state $\pi$ (figure 14(b)). Any violation of the assumption causes the appearance of state...
π in the global LTS. If this is the case, the error detection mechanism would indicate the transitions at which violation occurs. In the case of GasStation, we find that the global LTS remains the same as that in figure 13 after the inclusion of the LTS in figure 14(b). Since the global LTS is free of state π, figure 14(a) is a valid (supported) assumption.

Using the same philosophy, we can express various safety properties as user-specified interface processes and include them in the contextual reachability analysis. Violation of these properties is flagged as the appearance of state π in the global LTS. The error detection mechanism can then identify those safety properties being violated.

5. Conclusion

This paper presents a technique that enables users to exploit their additional knowledge of the behaviour of a distributed finite-state system when performing behaviour analysis. The additional knowledge, specified as interface processes, helps to reduce the computational costs in contextual reachability analysis. However knowledge specified by users could be incorrect and affect the correctness of the analysis result. An error detection mechanism is introduced to check the correctness. This is achieved by augmenting the model of labelled transition systems (LTS) with a special undefined state π. A process is said to be undefined when it is in state π. The analysis result is incorrect if the global LTS representing the overall system contains the state π. The error detection mechanism is effective in the sense that it does not increase the order of computational complexity of the contextual reachability analysis. The utilisation of the mechanism only increases the size of subsystem by at most one state to accommodate state π.

The error detection mechanism incorporates procedures to identify erroneous interface processes and missing transitions. Interface processes supplied by users can be combined directly with those derived algorithmically for behaviour analysis. The combination of these two sources of context constraints facilitates an effective contextual reachability analysis which can
be used to support the design activities of large-scale distributed systems. Assumptions about the behaviour of environment made during the design of subsystems can be validated during the construction of the global LTS.

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 [9]. We are also experimenting a framework to integrate the described technique with a dataflow analysis technique [2, 3].

References


[16] W. J. Yeh, “Controlling State Explosion in Reachability Analysis,” SERC, Purdue University, December 1993.