The TRACTA Approach for Behaviour Analysis of Concurrent Systems

Imperial College Research Report No. DoC 95/16

Dimitra Giannakopoulou

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London, SW7 2BZ, UK
Email: dg1@doc.ic.ac.uk

12 September 1995

Abstract

The need for modularity in the behaviour analysis of concurrent systems has been answered successfully by making reachability analysis compositional. Compositional reachability analysis (CRA) on the other hand, often exacerbates the state explosion problem; subsystem analysis leaves out information from the subsystem environment (context), which could considerably reduce the number of states allowed into its behaviour state-graph. To deal with that, we have chosen to incorporate context constraints in CRA. In the TRACTA approach developed in our section, context constraints are expressed as processes in our model (we call them *interface processes*), that are composed with the subsystem, without affecting the global system behaviour. TRACTA supports both automatically generated and user-specified interfaces. It also provides an elegant way of checking violation of safety properties by the system under analysis. This work, besides introducing the main open problems in this area of research, is a detailed presentation of TRACTA and its underlying theory, in their current form.

Keywords

compositional reachability analysis, state explosion problem, concurrent systems, labelled transition systems, context constraints.

1 Introduction

Behaviour analysis is useful at all stages in the software life cycle. Its main objective is to check if a system conforms to its requirements. The larger and more complex a software product, the more difficult it is to "manually" check that it operates as specified. The need is therefore intense for software analysis techniques that can be supported by efficient automated tools.

Most concurrent systems interact continuously with the environment and often do not terminate. Therefore, their behaviour cannot be adequately described by an input-output relation (*reactive* systems). On the other hand, they are inherently non-deterministic; the order in which events will occur is usually unpredictable and is only restricted by the synchronisation parts of the behaviour of processes. Concurrent systems are thus differentiated from their sequential counterparts and behaviour analysis techniques have to be specifically developed to cope with their special requirements.

Behaviour analysis techniques for concurrent systems are usually classified as:

Dynamic. Dynamic analysis techniques consist of testing a program on a set of executions. They focus on defining when a set of executions can be characterised as "complete". For the case of concurrent systems, they often additionally control the non-deterministic occurrence of events by introducing synchronisation points to processes, in order to force events to happen in predefined sequences. Dynamic analysis can only be used to show the presence of errors but cannot guarantee correctness in the general case. This is due to the fact that it can only cover a finite number of cases.

Static. Static analysis does not require the execution of the program. It can be used to show the absence of faults based on an abstract model of the implementation. More precisely, two classes of system properties can be verified using static analysis, *safety* and *liveness* properties. Safety properties assert that a program never enters an undesirable state (e.g. violation of mutual exclusion), whereas liveness properties assert that a program eventually enters a desirable state (e.g. freedom from starvation) [1, 2].

A program should, of course, be analysed by using a combination of the two techniques. This would involve, at the design stage, the verification of safety and liveness properties using some static analysis technique on an abstract model of the program. After a running implementation of the program has been created, the program may be executed for a number of cases, as required by some dynamic analysis technique.

In this work, we concentrate on building efficient static analysis techniques. The remainder of the report is organised as follows: section 2 discusses the main requirements from static analysis techniques and existing approaches towards satisfying these requirements. In section 3, reachability analysis is presented as a technique for performing static analysis. The state explosion problem as related to reachability analysis is then analysed and compositional reachability analysis and various issues that it raises are introduced. In section 4 the model used by the TRACTA technique is described. The technique itself is presented in section 5. Our conclusions and plans for future work are finally included in section 6. For better readability of the report, all proofs of theorems and lemmas are omitted from its main part, but can be found in appendix A.

2 Static analysis

The basic features of interest when building a static analysis technique are the following:

- A formal notation for specifying concurrent systems.
- A formal notation for specifying properties required for a concurrent system.
- An analysis technique that can be supported by an efficient automated tool.

These features are described in more detail in the following sections. Section 2.1 focuses on the modelling of systems and properties, whereas section 2.2 establishes the general requirements for analysis techniques. Finally, section 2.3 presents a selection of existing approaches to analysing concurrent systems.

2.1 Modelling of systems and their properties

A computational model is required to provide a formal notation for specifying concurrent systems. A number of models have been proposed, each of them capturing a particular type of semantic property of concurrent systems. Such models fall into two categories: those taking a *logical* and those taking a *behavioural* approach [2]. In the former, one describes the behaviour of a system by a set of axioms written in a formal notation such as temporal logic. System behaviour is defined to be the maximal set of event sequences that satisfy these axioms. In the behavioural approach, system behaviour is described by an abstract program, written in terms of some formal model such as regular expressions, automata or state transition diagrams. We believe that the behavioural approach is closer to a user's view of the system. Formal specification of a system in terms of a behavioural model should therefore be more straightforward.

Concurrent systems usually consist of various subprograms (processes) which run in parallel. A composition operator is therefore a necessary element of a behavioural model for concurrent systems. The user will thus be able to model a system in terms of its elementary component processes which, when composed by means of such an operator, express the global behaviour of the system (see Milner's *CCS* operator | in [11], and Hoare's *CSP* operator || in [10], for example).

A desirable feature from such models is that they have a graphical notation and are familiar to the user. As far as the specification of properties is concerned, the model needs to be rich enough to express the classes of properties that are of interest. Note here that the model for specifying the system behaviour does not need to be the same as the one for specifying the properties required from the system, as can be seen in section 2.3.

2.1.1 Limitations

Any model for specifying a system and/or its properties is bound to have inherent limitations [2], more precisely:

- it can only cover some aspects of the system behaviour;
- there are limitations in the correspondence between formal descriptions and the real world;

- there exists yet no widely accepted formal notation for modelling non-functional properties like performance, reliability, maintainability and availability.

2.2 Analysis of concurrent systems

When building an analysis technique, one has to take into account both that the following: "Does program P obey specification S" and that the presence of faults are undecidable properties for arbitrary programs and specifications [15, 13]. This implies that every behaviour analysis technique embodies some compromise between accuracy and computational costs. Moreover, there is no single technique capable of addressing all fault-detection concerns for arbitrary programs.

We will therefore have to restrict our work to deal with specific classes of problems. Firstly, we decide to focus on the synchronisation structures of concurrent systems. Data values in program execution are therefore abstracted away, either because they are of no consequence, or because removing them makes the model easier to analyse. Secondly, we will only consider finite-state concurrent systems. This approach is of wide applicability, since a large number of concurrent programming problems have finite-state solutions [7]. Having thus defined the classes of problems that we expect an analysis method to address, we will proceed by setting some requirements for the method itself [2].

It is essential for any behaviour analysis technique to have a sound underlying theory. The success of the technique will then depend on how well it can be supported by an automated tool. This is because the main objective of research in this area is to avoid manual verification of software, especially for concurrent systems where this is practically impossible due to their inherent complexity.

Recent concurrent systems are mostly developed in a compositional way. This involves a top-down decomposition of a system into a hierarchy of simpler components and a bottom-up composition of the system starting from its primitive components [2]. An analysis technique should therefore support compositionality, in order to allow system developers to analyse their designs incrementally according to some compositional hierarchy.

The technique should finally be general enough to handle both safety and liveness properties.

2.3 Existing approaches to static analysis

In the traditional approach to concurrent program analysis (logical approach), a system is described by a set of axioms written in a formal notation (usually temporal logic), and the proof that the system meets its specification is constructed using these axioms together with a set of inference rules in this logic. The task of proof construction is in general quite tedious, and a good deal of ingenuity may be required to organise the proof in a manageable way. Even worse, automatic theorem provers have failed to be of much help due to the inherent complexity of testing validity even for the simplest logics [7]. Finally, it is very difficult to express a complex concurrent system in terms of axioms in some logic, which would discourage software developers from using it.

It is obvious that a technique becomes more appealing if it allows the developers to specify the primitive processes which, when running in parallel, make up the concurrent system that they wish to analyse. Such techniques require a composition operator, that captures parallelism in its semantics. The mechanism of composing processes related to each other

by means of a composition operator should be supported by an efficient automated tool. The possibility of automatically synthesising finite-state concurrent systems from temporal logic specifications has been considered, but the synthesis algorithms have exponential time complexity in the worst case [7].

Reachability analysis is a widely used technique for exhaustive analysis of finite-state concurrent systems. It involves the construction of a global-state graph of the system (also referred to as reachability graph) by composing state-graphs of its primitive components. Reachability analysis is an attractive technique because it is simple and relatively easy to automate [2]. Various methods have been proposed for verifying properties when the global-state graph of the system is available.

In [7], Clarke et al. present a branching-time temporal logic, *CTL* (Computation Tree Logic), for specifying properties, as well as an efficient model checker for verifying *CTL* formulae against the global-state graph of the system. The expressiveness of *CTL* has been restricted for the benefits of efficiency. The main advantage of the method lies in its flexibility; it provides a uniform notation for expressing a wide variety of correctness properties. Moreover, the model checker can handle both safety and liveness properties with equal facility.

On the global state-graph of a system, deadlock states are easily detected as the states with no outgoing transitions. Our analysis technique, TRACTA, additionally provides an elegant method for checking safety properties [6]. The method consists of composing into the system "image automata" that correspond to safety properties. The advantages of the method are that: 1) properties can be expressed using a graphical notation, 2) properties may contain actions that are not globally observable, 3) violation of properties is detected by undefined states in the global-state graph, which implies that no additional work is needed after the construction of the graph, 4) the method also locates the violation in the property automaton. The method is presented in detail later on.

3 Reachability analysis

Reachability analysis is a group of concurrency analysis techniques involving enumeration of all reachable states in a finite-state model. The composite state-transition model, often called a *reachability graph*, is constructed from models of individual processes abstracted from the system being analysed, and contains all reachable states in the system. The reachability graph is then analysed for general properties like freedom from deadlocks and livelocks [14].

The use of a composition operator is indispensable in the construction of the composite state-transition model of the system from models of its individual components. Reachability analysis can therefore be applied to models like Petri nets, *CCS*, or *CSP* [12, 11, 10]. We will simplify our discussion by using the *CSP* process algebra for specifying state graphs, and the *CSP* composition operator $\|$ to connect individual processes that run in parallel. Given the close correspondence between *CSP* and *CCS*, the problems that we will present below also apply for the case of *CCS*.

The problem that needs solving when performing simple reachability analysis stems from the interleaving semantics of the composition operator; two processes are composed by synchronisation of the actions common to both their alphabets, and interleaving of the others. More precisely, let us assume that we are composing n *CSP* processes $P_1, P_2, \ldots, P_n$ with numbers of states $s_1, s_2, \ldots, s_n$ respectively. Let the states of the composite process P be named by using n digits, the first one to denote the state of P_1 in the composite process, the

second one to denote the state of P_2, and so on. In the case where all combinations should be possible (worst case), system P would consist of $s_1 \times s_2 \times \cdots \times s_n$ states. Reachability analysis has therefore exponential complexity, which is often referred to as the *state explosion problem* [14, 2].
There exist two factors that may help to reduce the size of the problem:

1. arbitrary interleaving is pruned out by the need to synchronise actions common in the process alphabets;

2. by use of the hiding operator. Actions which are internal in subsystems or which the developer does not want observable in the global-state graph because they are of no interest, can be turned into the unobservable τ action. The composite graph can then be minimised in terms of the equivalence that is of interest to the user.

As far as the second factor is concerned, if composition of the system is performed in one step as in conventional reachability analysis, hiding can only be applied after the global graph has been computed. In this way, if the state explosion problem is to occur, it will when constructing the reachability graph, i.e. well before hiding is attempted.

An appealing way for tackling this problem has been to introduce compositionality into conventional reachability analysis. A useful compositional reachability analysis technique must support a divide-and-conquer strategy [14] so that reachability graph representations of subsystems can be independently derived and simplified, and then hierarchically combined to form representations of successively larger parts of the complete system. A compositional approach would also be incremental, since changes to one subsystem would not invalidate the reachability graph representation of the other.

Although Compositional Reachability Analysis (CRA) may control the state explosion problem in some cases, it will not significantly reduce its size in the worst case. In fact, the problem may even be exacerbated as subsystems are composed without using synchronisation information from their environment (context).

In order to deal with this problem often referred to as the *intermediate state explosion problem* [14], Graf and Steffen have proposed to include in the composition of subsystems, user-specified interfaces which reflect the user's intuition about the behaviour of the subsystem environment [9]. They have extended finite state transition systems by the undefinedness predicate $\uparrow$. A state transition system is totally defined if its undefinedness predicate is empty. Inclusion of erroneous user-specified interfaces in the CRA of some system is revealed in the the resulting global transition system by the fact that it is not totally defined. The proposed method for including constraints in CRA is very interesting. What it lacks is an automatic way of producing such interfaces, to make the task of defining interfaces optional to the user.

In his PhD thesis [14] Yeh proposes a *SLEEP/WAKE/ACTIVATE* mechanism according to which developers may introduce *SLEEP/WAKE/ACTIVATE* transitions in the transition systems of processes, in order to express their intuition concerning constraints imposed to these processes by their context. The main disadvantage of this method is that it does not preserve the associativity of the composition operator. Moreover, as for the case of [9], all the burden of specifying context-constraints remains with the developer of the system.

We have modified Graf and Steffen's proposed method of capturing context constraints into user-specified interfaces. Rather than first composing processes together and subsequently adding their undefined transitions, interface processes are substituted by their "im-

age" equivalents which are then normally composed into the system. Moreover, we provide an algorithm for automatically generating interfaces that express context constraints of subsystems. The algorithm provides an effective and practical means of constructing interfaces from a context composed of small or medium size elementary processes [2].

4 The model

4.1 Labelled Transition Systems

The behaviour of a synchronous communicating process in a distributed system can be modelled as a rooted *Labelled Transition System (LTS)*. Informally, the *LTS* of a process is a state transition diagram containing all its reachable states and executable transitions.

The set of actions that are considered relevant for a particular description of a process P is called its communicating alphabet, written as αP. Processes with different alphabets are distinguished, even if they behave the same [10]. It is therefore important to attach an alphabet to each process and to declare this explicitly. It is impossible for a process to perform an action outside its alphabet. The choice of an alphabet can thus be related to a deliberate simplification to make analysis practical. This simplification involves decisions to ignore many other properties and actions considered to be of lesser interest.

Let us now introduce some of the notation that will be used in the presentation of the model:

- $\mathcal{A}$ is the universal set of observable actions.
- *Act* equals $(\mathcal{A} \cup \tau)$, where τ is the internal unobservable action.
- Σ is the universal set of states, not containing state π. State π is a special state introduced in our model, called the *undefined state*. The properties and usage of π are described further on.
- *Sts* equals $(\Sigma \cup \pi)$.

Unless otherwise specified, actions $a, b, c, \ldots$ will range over $\mathcal{A}$, actions $\alpha, \beta, \gamma, \ldots$ will range over Act, whereas $s, t, \ldots$ will be action sequences ranging over Act^*.

In this model we have modified the traditional definition of *LTS* to include the undefined state π [9, 3]. We therefore formally define an *LTS* as follows:

Definition 4.1 *An* LTS *is a triple* $\langle S, A, \Delta \rangle$*, where:*

(i) $S \subseteq$ Sts *is a finite set of states;*

(ii) $A = A' \cup \{\tau\}$*, where* $A' \subseteq \mathcal{A}$ *is a set of observable actions;*

(iii) $\Delta \subseteq (S - \{\pi\}) \times A \times S$ *is a transition relation*[1]*. A transition* $\langle p, \alpha, q \rangle$ *will also be written as* $p \xrightarrow{\alpha} q$*.* □

[1] Note that state π is not allowed to have outgoing transitions.

To every transition relation $\Delta \subseteq (S - \{\pi\}) \times A \times S$ corresponds a single function $f_\Delta : ((S - \{\pi\}) \times A) \longrightarrow 2^S$ defined as follows (a similar definition exists in [8]):

$$\forall p \in (S - \{\pi\}), \quad f_\Delta(p, \alpha) = \begin{cases} \{q \mid \langle p, \alpha, q\rangle \in \Delta\}, & \text{for } \alpha \in A \\ \{p\}, & \text{otherwise.} \end{cases}$$

In the same way, a function $f_\Delta : ((S - \{\pi\}) \times A) \longrightarrow 2^S$ defines a single transition relation $\Delta \subseteq (S - \{\pi\}) \times A \times S$.

Definition 4.2 *Given an LTS $T = \langle S, A, \Delta\rangle$ and states $q, r \in S$, we say that state r is reachable from state q iff $(q = r)$ or $\exists(\alpha_1, \ldots, \alpha_n, q_1, \ldots, q_{n-1})$ with $(\alpha_1, \ldots, \alpha_n \in A, q_1, \ldots, q_{n-1} \in S, n \geq 1)$, such that $(\forall i,\ 0 \leq i < n,\ q_i \stackrel{\alpha_{i+1}}{\longrightarrow} q_{i+1} \in \Delta)$, where $(q_0 = q, q_n = r)$. We then say that state r is reachable from state q by the sequence of actions $t = \alpha_1 \cdots \alpha_n$.* □

Definition 4.3 *A process P is a quadruple $\langle S_p, A_p, \Delta_p, p\rangle$ formed by an LTS $T = \langle S, A, \Delta\rangle$ and a designated initial state $p \in S$ in the following way:*

(i) S_p is the set of states that are reachable from p in T;

(ii) $A_p = A$. We call $A' = A_p - \{\tau\}$ the communicating alphabet of P, denoted as αP;

(iii) $\Delta_p = \{\langle q, \alpha, r\rangle \mid (\langle q, \alpha, r\rangle \in \Delta) \wedge (q, r \in S_p)\}$. □

We identify each state $p \neq \pi$ of an *LTS* T with the process P formed by T with initial state p. In any *LTS* T, state π is identified with process $\Pi = \langle\{\pi\}, Act, \emptyset, \pi\rangle$, called the *undefined* process.

Definition 4.4 *We say that $P = \langle S, A, \Delta, p\rangle$ transits into $P' = \langle S', A', \Delta', p'\rangle$ with an action α and write $P \stackrel{\alpha}{\longrightarrow} P'$ iff $\langle p, \alpha, p'\rangle \in \Delta$ and P' is the process that is identified with state p' in $\langle S, A, \Delta\rangle$. In general, given an action sequence $t \in Act^*$ where $t = \alpha_1 \cdots \alpha_n$,*

$$P \stackrel{t}{\longrightarrow} P' \text{ iff } P \stackrel{\alpha_1}{\longrightarrow} \cdots \stackrel{\alpha_n}{\longrightarrow} P'.$$ □

Definition 4.5 *If $t = \alpha_1 \cdots \alpha_n \in Act^*$, then $P \stackrel{t}{\Longrightarrow} P'$ iff*

$$P(\stackrel{\tau^*}{\longrightarrow}) \stackrel{\alpha_1}{\longrightarrow} (\stackrel{\tau^*}{\longrightarrow}) \cdots (\stackrel{\tau^*}{\longrightarrow}) \stackrel{\alpha_n}{\longrightarrow} (\stackrel{\tau^*}{\longrightarrow})P'.$$ □

Therefore $P \stackrel{a}{\Longrightarrow} P'$ means that $P \stackrel{\tau^m}{\longrightarrow}\stackrel{a}{\longrightarrow}\stackrel{\tau^n}{\longrightarrow} P'$ for some $m, n \geq 0$. Similarly, $P \stackrel{\varepsilon}{\Longrightarrow} P'$ iff $P \stackrel{\tau^m}{\longrightarrow} P'$ for some $m \geq 0$.

Definition 4.6 *A process $P = \langle S, A, \Delta, p\rangle$ is said to be deterministic iff*

$$\forall s, s', s'' \in S,\ (\langle s, \alpha, s'\rangle \in \Delta \ \wedge\ \langle s, \alpha, s''\rangle \in \Delta) \Rightarrow s' = s'';$$

otherwise it is said to be non-deterministic.

A *trace* of a process P is a sequence of observable actions that P can perform starting from its initial state (see [10] for formal definition). We denote the set of possible traces of a process P as $tr(P)$. A trace t of a process P is said to be *undefined* iff $P \stackrel{t}{\Longrightarrow} \Pi$. A process is *totally defined* iff it does not contain undefined traces.

An important operation on traces is catenation [10], which constructs a trace from a pair of operands s and t by simply putting them together in this order; the result will be denoted $s^\wedge t$.

4.2 Operators

As in conventional reachability analysis, our method makes use of the *hiding* and the *composition* operators. The hiding operator $\uparrow$ takes two arguments, a process $P = \langle S, A, \Delta, q\rangle$ and a set of observable actions L and returns process $P \uparrow L$. $P \uparrow L$ is the process projected from P in which only the actions in L are observable, whereas actions in $A - L$ are turned into τ. Definition 4.7 is a formal definition of the hiding operator. A more intuitive way of describing the hiding operator is in terms of its transitional semantics, as is illustrated in table 1 (rules 1–2).

Definition 4.7 *Process $P \uparrow L$ where $P = \langle S, A, \Delta, p\rangle$ and L is a set of observable actions, equals process $\langle S', A', \Delta', p\rangle$, where*

(i) $S' = S$;

(ii) $A' = (A \cap L) \cup \{\tau\}$;

(iii) $\Delta' = \{\langle p, \alpha, q\rangle \mid \langle p, \alpha, q\rangle \in \Delta, \alpha \in L\} \cup \{\langle p, \tau, q\rangle \mid \langle p, \alpha, q\rangle \in \Delta, \alpha \in (A - L)\}$. □

The following lemma (proof is trivial given the transition semantics of the hiding operator) states an important property of the hiding operator:

Lemma 4.1 *For any process P, and any set of observable actions L, P has undefined traces iff $P \uparrow L$ has undefined traces.* □

The composition operator is a binary operator taking as arguments two *LTS* or two processes. In order to define composition among processes, we have first defined composition among *LTS* as follows (definition 4.9):

Definition 4.8 *Let $R, S \subseteq Sts$ be two sets of states. We define $R \otimes S$ in the following way:*

$$R \otimes S = \begin{cases} ((R - \{\pi\}) \times (S - \{\pi\})) \cup \{\pi\}, & \text{if } (\pi \in R) \vee (\pi \in S) \\ R \times S, & \text{otherwise.} \end{cases}$$

□

Definition 4.9 *Let $T_1 = \langle S_1, A_1, \Delta_1\rangle$ and $T_2 = \langle S_2, A_2, \Delta_2\rangle$ be two LTS. Their composition $T_1 \parallel T_2$ is a labelled transition system $\langle S, A, \Delta\rangle$, where*

(i) $S = S_1 \otimes S_2$;

(ii) $A = A_1 \cup A_2$;

(iii) Δ *is the transition relation that corresponds to function* $f_\Delta : ((S - \{\pi\}) \times A) \longrightarrow 2^S$ *defined as follows:*

$\forall (p, p') \in S,\ \forall \alpha \in A,$

$$f_\Delta((p, p'), \alpha) = \begin{cases} f_{\Delta_1}(p, \alpha) \otimes f_{\Delta_2}(p', \alpha), & \text{for } \alpha \neq \tau \\ (\{p\} \otimes f_{\Delta_2}(p', \alpha)) \cup (f_{\Delta_1}(p, \alpha) \otimes \{p'\}), & \text{otherwise.} \end{cases}$$

□

The notion of composition used in this definition adopts the style of *CSP*, i.e. synchronisation of the actions common to the alphabets of the component *LTS* and interleaving of the others. We can now proceed to the formal definition of composition for processes.

Definition 4.10 *Let* $P_1 = \langle S_1, A_1, \Delta_1, p_1 \rangle$ *and* $P_2 = \langle S_2, A_2, \Delta_2, p_2 \rangle$ *be two processes, and* $T = \langle S, A, \Delta \rangle$ *be the* LTS *obtained from the composition* $\langle S_1, A_1, \Delta_1 \rangle \parallel \langle S_2, A_2, \Delta_2 \rangle$ *of their corresponding* LTSs. *Then the composition* $P_1 \parallel P_2$ *of the two processes is defined to be the process that is formed from* T *with initial state* (p_1, p_2) *if* $(p_1, p_2 \neq \pi)$, *and process* Π *otherwise.*

For a simpler description of the way the composition operator works, we have included its transitional semantics in table 1 (rules 3–5). The composition operator is both commutative and associative [10], and therefore allows for any number of processes to be composed in any order. This property is very important when performing compositional reachability analysis, because all the compositional hierarchies of a set of processes will produce equivalent results.

The alphabet A of a composite process $P \parallel Q$ has to be $\alpha P \cup \alpha Q$, even if $P \parallel Q$ cannot actually perform all of the actions in $\alpha P \cup \alpha Q$. Associativity of the composition operator would not be preserved if the alphabet of the composite process was defined to be the set of actions that it can actually perform, as is depicted in figure 1.

Hiding operator ($\uparrow$)			
1a.	$\frac{P \xrightarrow{a} P'}{P\uparrow L \xrightarrow{a} P'\uparrow L} \quad (a \in L, P' \neq \Pi)$	1b.	$\frac{P \xrightarrow{a} \Pi}{P\uparrow L \xrightarrow{a} \Pi} \quad (a \in L)$
2a.	$\frac{P \xrightarrow{a} P'}{P\uparrow L \xrightarrow{\tau} P'\uparrow L} \quad (a \notin L, P' \neq \Pi)$	2b.	$\frac{P \xrightarrow{a} \Pi}{P\uparrow L \xrightarrow{\tau} \Pi} \quad (a \notin L)$
Composition operator ($\parallel$)			
3a.	$\frac{P \xrightarrow{a} P'}{P\|\|Q \xrightarrow{a} P'\|\|Q} \quad (a \notin \alpha Q, P' \neq \Pi)$	3b.	$\frac{P \xrightarrow{a} \Pi}{P\|\|Q \xrightarrow{a} \Pi} \quad (a \notin \alpha Q)$
4a.	$\frac{Q \xrightarrow{a} Q'}{P\|\|Q \xrightarrow{a} P\|\|Q'} \quad (a \notin \alpha P, Q' \neq \Pi)$	4b.	$\frac{Q \xrightarrow{a} \Pi}{P\|\|Q \xrightarrow{a} \Pi} \quad (a \notin \alpha P)$
5a.	$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P\|\|Q \xrightarrow{a} P'\|\|Q'} \quad (a \neq \tau, P' \neq \Pi, Q' \neq \Pi)$		
5b.	$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P\|\|Q \xrightarrow{a} \Pi} \quad (a \neq \tau, (P' = \Pi \vee Q' = \Pi))$		

Table 1: Transitional semantics for the $\uparrow$ and $\parallel$ operators. Here, action a can be a τ when the rules allow it.

4.3 Behavioural equivalences

Strong semantic equivalence, denoted as $\sim$, is used to relate two processes whose behaviours are indistinguishable to an observer, even if internal τ-actions are observable. *Weak semantic equivalence*, denoted as $\approx$, is used to relate two processes whose behaviours are indistinguishable to an observer if internal τ-actions are not observable. In this section, we extend Milner's definitions of strong and weak equivalence for *CCS* [11] to fit the needs of our model (definition 4.11 is a preliminary definition [10]).

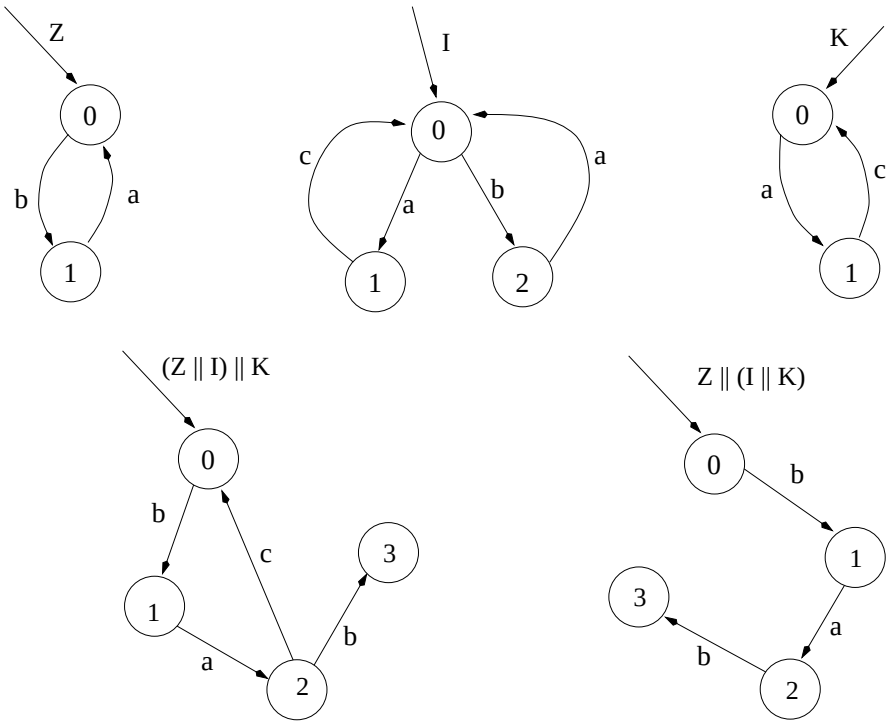

Figure 1: Associativity of the composition operator is not preserved if the alphabet of the composite process is defined to be the set of actions that the process can actually perform. The result obtained for $(Z \parallel I \parallel K)$ by using the correct definition is identical to $Z \parallel (I \parallel K)$ in the figure above.

Definition 4.11 *If $t \in Act^*$, then $\widehat{t} \in \mathcal{A}^*$ is the sequence gained by deleting all occurrences of τ from t.* □

Note, in particular, that $\widehat{\tau^n} = \varepsilon$ (the empty sequence).

Definition 4.12 *A strong semantic equivalence $\sim$ is the union of all relations $\boldsymbol{R} \subseteq Sts \times Sts$, called strong bisimulation relations, where $(P, Q) \in \boldsymbol{R}$ implies:*

1. $\alpha P = \alpha Q$;
2. $P = \Pi$ *iff* $Q = \Pi$;
3. $\forall \alpha \in Act$,
   - *(i)* $P \xrightarrow{\alpha} P'$ *implies* $\exists Q', Q \xrightarrow{\alpha} Q'$ *and* $(P', Q') \in \boldsymbol{R}$;
   - *(ii)* $Q \xrightarrow{\alpha} Q'$ *implies* $\exists P', P \xrightarrow{\alpha} P'$ *and* $(P', Q') \in \boldsymbol{R}$. □

Definition 4.13 *A weak semantic equivalence $\approx$ is the union of all relations $\boldsymbol{R} \subseteq Sts \times Sts$, called weak bisimulation relations, where $(P, Q) \in \boldsymbol{R}$ implies:*

1. $\alpha P = \alpha Q$;
2. $P = \Pi$ *iff* $Q = \Pi$;
3. $\forall \alpha \in Act$,
   - *(i)* $P \xrightarrow{\alpha} P'$ *implies* $\exists Q', Q \stackrel{\widehat{\alpha}}{\Longrightarrow} Q'$ *and* $(P', Q') \in \boldsymbol{R}$;
   - *(ii)* $Q \xrightarrow{\alpha} Q'$ *implies* $\exists P', P \stackrel{\widehat{\alpha}}{\Longrightarrow} P'$ *and* $(P', Q') \in \boldsymbol{R}$. □

We have adapted to our model Milner's definitions of strong and weak equivalences for *CCS* [11]. More precisely, bisimilar processes need to have the same alphabets for the composition operator to preserve bisimilarity[2] (first rule, definitions 4.12, 4.13). We have also extended the definitions to include the undefined process Π (second rule, definitions 4.12, 4.13).

5 The TRACTA technique

Compositional reachability analysis (CRA) is performed on a system that is expressed as a hierarchical division of subsystems. It is therefore guided by a compositional hierarchy of subsystems, where the root of the hierarchy is a system S to be analysed, and the leaves of the hierarchy are the primitive components C_i of S. Although different compositional hierarchies may increase or reduce the size of the state explosion problem, they all return equivalent results since the composition operator is both commutative and associative. The choice of a specific hierarchy is usually related to the natural division of a system into subsystems.

The behaviour of each primitive component C_i is specified in terms of a pair consisting of an *LTS* L_i and an initial state s_i. The reachability graph for C_i is described by the process[3] P_i that is identified, in our model, with state s_i of L_i. We have to mention here that all processes

[2] The semantics of the composition operator in our model are similar to those of the *CSP* composition operator.

[3] We will from now on use the terms reachability graph and process interchangeably.

that describe primitive components of subsystems are totally defined. The undefined state is never used in the specification of a process. Its only use is in the verification of properties and of interface correctness, as described in sections 5.3 and 5.4.

The reachability graph of the global system S is composed step by step from those of its subsystems in a bottom-up manner [2], according to the compositional hierarchy. At each intermediate step, the resulting graph of each subsystem is simplified by hiding actions that are internal to the subsystem. CRA therefore consists of gradually performing compositions and simplifications of processes in our model. The final graph may then be used for verifying safety and liveness properties of the system under analysis.

Besides being incremental, the CRA technique may in some cases reduce the size of the state explosion problem by hiding from the analysis process details that are internal to subsystems. However, as discussed in section 3, it may in other cases trigger the intermediate state explosion problem. The latter appears when subsystems are locally composed without the use of synchronisation information from their environment (from now on referred to as context information[4]).

We have adopted the approach of using interface processes for including context information into CRA [9, 2]. We present our criteria for correctness of interface processes, as well as a technique for checking arbitrary interfaces against these criteria in section 5.3. We also describe a technique for validating safety properties in section 5.4. But before that, we discuss in section 5.1 some issues related to the use of the hiding operator in reachability analysis. Moreover, in section 5.2, we define the concept of an "image" process and present two theorems that are essential for the remainder of the paper.

5.1 The hiding operator

Rule 5.1 *For any two processes P, Q and two sets of actions L, M:*

$$P \uparrow L \parallel Q \uparrow M = (P \parallel Q) \uparrow (L \cup M) \text{ iff } ((\alpha P - L) \cap (\alpha Q \cup M)) \cup ((\alpha Q - M) \cap (\alpha P \cup L)) = \emptyset.$$

□

A proof for the rule is included in appendix A.

Without loss of generality, action renaming can be performed in such a way that the name of an action in any component of the system is globally unique unless it needs to synchronise with actions in other component processes of the system. Actions in distinct components will be given the same name if and only if they must synchronise. In this way, rule 5.1 is applicable at any level of some compositional hierarchy.

Given the above and the associativity of the $\parallel$ operator, we can show that, even in the presence of hiding, compositional reachability analysis is equivalent to conventional reachability analysis. Let $P_1, \ldots, P_n$ be the primitive processes of a system S. If hiding has occurred at some stages of CRA according to sets $S_1, \ldots, S_m$, then it is obvious that the global graph obtained will be equivalent to $(P_1 \parallel \ldots \parallel P_n) \uparrow (S_1 \cup \ldots \cup S_m)$, which is what conventional reachability analysis computes.

As discussed in section 3, CRA is much more effective in exploiting hidden actions than conventional reachability analysis. Additionally, it exhibits the following advantage: it is more convenient and intuitive to the user to hide actions according to his view of the system hierarchy, i.e. in terms of subsystems, rather than in terms of the whole system.

[4] In a system $S = P \parallel Q$ where P, Q may be decomposable processes (i.e. non-primitive subsystems), Q is the *context* of P and vice versa.

In the following sections, for simplifying the reasoning and proofs, we will assume that any system S can be expressed in the form $P \uparrow L$, where $P = (P_1 \parallel \ldots \parallel P_n)$ is a process in our model, and $L = (L_1 \cup \ldots \cup L_m)$ is a set of observable actions. This is obviously acceptable given rule 5.1 and the way in which we have required action renaming to be performed.

5.2 Two essential theorems

Definition 5.1 *Let $P = \langle A, S, \Delta, p\rangle$ be a totally defined process. Then we call image process of P the process $P' = \langle A, S \cup \{\pi\}, \Delta', p\rangle$, where Δ' is constructed from Δ by the following procedure [3]:*

(i) initialise Δ' to Δ;

(ii) for all $a \in A$ and $s \in S$ where there does not exist $s' \in S$ such that $\langle s, a, s'\rangle \in \Delta$: add $\langle s, a, \pi\rangle$ to Δ'.

Theorem 5.1 (Transparence theorem) *Let Z, P be two totally defined processes, where P is deterministic and free of internal action τ and $\sim$ denotes the strong equivalence relation. Then $Z \sim (Z \parallel P)$ iff:*

1. $\alpha P \subseteq \alpha Z$;
2. $tr(Z \uparrow \alpha P) \subseteq tr(P)$. □

Theorem 5.2 (Image process theorem) *Let P, T be two totally defined processes, and T' be the image process of T, where $\alpha T \subseteq \alpha P$. Then $P \parallel T'$ does not have undefined traces iff $tr(P \uparrow \alpha T) \subseteq tr(T)$.* □

The transparence theorem is an extended version of the interface theorem as presented in [4]. The image process theorem has been stated and used in [3, 6]. The full presentations and proofs of theorems 5.1 and 5.2 are provided in appendix A.

5.3 Inclusion of context constraints

Let S be a system $P \parallel Q$ where Q is the context of subsystem P. Then an interface I for P is a totally defined process with the following essential characteristic: I describes some part of Q's behaviour which, when composed with P will reduce (by the requirement for synchronisation) its reachability graph. It is therefore obvious that $\alpha I \subseteq \alpha P \cap \alpha Q$. The reason for introducing process I in the analysis is for exploiting context information in order to alleviate the intermediate state explosion problem. However, we do not wish this process to alter in any way the behaviour of the system S to be analysed. Therefore, supposing that eq is an equivalence relation that preserves the class of properties that are of interest when performing analysis, I should satisfy the following requirement:

$$(P \parallel Q) \;\; eq \;\; ((P \parallel I) \parallel Q)$$

or due to the fact that $\parallel$ is both commutative and associative,

$$S \;\; eq \;\; (S \parallel I).$$

We choose to use strong equivalence in our technique. Strong equivalence is the strongest equivalence of interest in concurrency theory and preserves any properties of interest, including divergence which is not preserved by observational equivalence. We therefore conclude in the following formal definition [9, 2]:

Definition 5.2 *Let $S = R \uparrow L$ be a system, where $R = (P \parallel Q)$. Then*

(i) a totally defined process I is an interface for P iff $\alpha I \subseteq (\alpha P \cap \alpha Q)$ (Q is the context of P in S);

(ii) an interface I for P is correct iff I is transparent to the system, i.e. $R \sim R \parallel I$. □

For the benefits of simplicity, we will for the moment ignore the hiding operator. Let S be the system under analysis. A component process P for which interface I has been introduced is substituted by $P' = P \parallel I$ in the compositional hierarchy on which CRA will be based. Given that $\alpha I \subseteq \alpha P$, composing I with P can only prune out transitions of P that would otherwise be allowed at this stage of the analysis. Therefore, although interface processes are additional processes composed into the system, this can only be for the best with respect to reducing the size of the state explosion problem.

When incorrect interfaces have been composed into the system, the reachability graph of the global system is not equivalent to the one that would have been obtained if the interface processes had been omitted. Therefore the final reachability graph obtained from CRA is useful for verifying properties in those cases only where all interfaces introduced into the analysis are correct. We thus need a way of verifying that $(S \parallel I) \sim S$, when the reachability graph of $S \parallel I$ is available but the one for S is not.

We have decided that it is reasonable to assume that only deterministic interfaces free of internal action τ will be introduced into the analysis of any system (the reason for our choice is provided in the proof of the transparence theorem, appendix A, page 23). We have been led to the same decision by the fact that checking correctness of interface I by verifying that $(S \parallel I) \sim S$ is out of the question for obvious reasons. We therefore would like to include in our methods interfaces that can be proven correct by the transparence theorem. So from now on, we will only refer to interfaces that are deterministic and free of τ.

According to the transparence theorem, a deterministic, free of action τ interface I is correct iff:

1. $\alpha I \subseteq \alpha S$;
2. $tr(S \uparrow \alpha I) \subseteq tr(I)$.

It is trivial to prove that the first condition is guaranteed by part 1 of definition 5.2 for any interface. Therefore interface I can be proven correct by checking that $tr(S \uparrow \alpha I) \subseteq tr(I)$.

Interfaces may be introduced in the analysis of a subsystem in two ways. Firstly they can be algorithmically derived. Algorithms designed to perform this task need to compromise accuracy of the interfaces they generate for efficiency. Even so, there exists yet no such algorithm that proves efficient in all cases. We have proposed a simple algorithm that is an effective and practical means of constructing interfaces from contexts composed of small or medium size elementary components [4]. The algorithm constructs interfaces that are deterministic and free of action τ. Interfaces created in this way have also proven to satisfy the

second condition of the transparence theorem. Every interface constructed by the algorithm is therefore guaranteed to be correct.

Secondly, interfaces can be specified by the users. In many situations, users have further knowledge and intuition of the system behaviour [3]. They may therefore be able to specify interfaces that capture the context constraints of subsystems. In this case however, any interface I introduced in the analysis of system S for some subsystem P where $S = P \parallel Q$, needs to be checked against the following conditions:

1. $\alpha I \subseteq (\alpha P \cap \alpha Q)$;
2. $tr(S \uparrow \alpha I) \subseteq tr(I)$;
3. I is deterministic free of action τ.

Conditions 1 and 3 are trivial to check. Condition 2, on the other hand, is not as straightforward to verify. The image process theorem simplifies the task considerably. For any interface process I that is introduced into the analysis, rather than composing I into S, we decide to compose its corresponding image process I'. Theorem 5.2 then guarantees that condition 2 of the theorem will be satisfied iff there are no undefined traces in $S \parallel I'$, i.e. iff state π does not appear in the global reachability graph.

The disadvantage of the method is that an incorrect interface I is detected only at the final stage of CRA. Obviously, the global reachability graph of the system obtained in the presence of incorrect interfaces may not be used for performing further analysis. That means that the reachability graphs of all subsystems for which I is a descendant in the compositional hierarchy will have to be recomputed. This can unfortunately not be avoided, because undefined traces that appear at intermediate stages of CRA may disappear at latter stages. We have to assume, however, that whenever users decide that they can help the analysis by introducing interfaces, these interfaces will in most cases be correct.

On the other hand, if the final graph does not contain undefined traces, it is guaranteed that all interfaces introduced are correct. For any interface I, the set $tr(I') - tr(I)$ contains exactly[5] those traces in I' that are undefined. So given that state π is not part of the reachability graph for the system under analysis, we conclude that $S \parallel I' \sim S \parallel I$. Therefore, for the case of correct interfaces, the result obtained from CRA is not affected by the fact that they have been substituted by their image equivalent in the compositional hierarchy.

We will now prove that our technique is not affected by the existence of the hiding operator in CRA. As we have mentioned in section 5.1, we can assume that any system S can be expressed in the form $P \uparrow L$. For some interface I, the reachability graph obtained by CRA is equivalent to $(P \parallel I') \uparrow L$. By lemma 4.1, if $(P \parallel I') \uparrow L$ does not have undefined traces neither does $P \parallel I'$. Therefore, by the image process and transparence theorems, $(P \parallel I) \sim P$. According to definition 5.2, the latter proves I correct. Moreover, since $P \parallel I'$ does not have undefined traces, $(P \parallel I') \sim (P \parallel I)$ which implies that $(P \parallel I') \uparrow L \sim (P \parallel I) \uparrow L$. The latter proves that for correct interfaces, the result obtained by composing their image processes into the system rather than themselves does not affect the final result, even in the presence of hiding.

[5] See definition 5.1, step 2.

5.4 Checking properties in CRA

The reachability graph of a system can be used for checking two classes of properties: liveness and safety properties. Deadlock in a reachability graph is related to the existence of deadlock states (definition 5.3) and is therefore easy to check.

Definition 5.3 *Let $P = \langle S, A, \Delta, p \rangle$ be a process. Then a state $s \in S$ is a deadlock state iff*

$$\forall \alpha \in A, \;\; f_\Delta(s, \alpha) = \emptyset.$$

□

A large number of safety properties can be expressed as processes in our model that are deterministic, free of internal action τ and of undefined traces. We will describe a technique for validating safety properties described in this way. The technique is based on the transparence theorem and the image process theorem.

Definition 5.4 *Let S be a process equals $Q \uparrow L$. Then a property T such that $\alpha T \subseteq \alpha S$ is violated by S iff $tr(Q \uparrow \alpha T) \not\subseteq tr(T)$.* □

For simplicity, let us for the moment ignore hidden actions and assume that we want to check if process S violates property T. By definition 5.4 and by the image process theorem, it is enough to check if $S \parallel T'$ has undefined traces, where T' is the image process of T.

Similar to the case for interfaces, image processes corresponding to the properties that the system has to satisfy are composed into the system. If the reachability graph obtained is free of undefined traces, then the system does not violate any of the properties in question.

When the global graph obtained has no undefined traces, the graph is strongly equivalent to the one that would have been obtained had the image processes for properties not been composed into the system. This is proven as follows: similar to the case for interfaces, $(S \parallel T') \sim (S \parallel T)$. Moreover, for every process T, $tr(S \uparrow \alpha T) \subseteq tr(T)$, and T is deterministic and free of τ and undefined traces. By the transparence theorem $(S \parallel T) \sim S$ and therefore $(S \parallel T') \sim (S \parallel T) \sim S$.

An issue that remains to be examined is where to locate an image process T' corresponding to some property T, in the compositional hierarchy of a system S. It is possible for a subsystem of S to violate property T when it is not in the context of S. This is the reason why, at some stage of the analysis, undefined states might occur which will not survive up to the final stage where the global reachability graph is obtained. We wish to avoid as many such "fake" undefined states as possible, therefore, we should place T' as high in the hierarchy as possible. There is an obvious restriction to the above[6]: for any node i in the compositional hierarchy such that $(\alpha T' \cap A_i \neq \emptyset)$, T' must be located at a descendant node of i in the tree of the hierarchy. A_i is used here to denote the set of observable actions that are hidden at the stage of CRA corresponding to node i.

One should therefore proceed as follows in order to specify a location for T' in the compositional hierarchy: find a subsystem S_i s.t. $\alpha T \subseteq \alpha S_i$, and then follow the path upwards until the restriction presented above is violated[7] or the root of the hierarchy is reached. It is possible for this procedure not to succeed in finding a location for T', in the single case where different actions of T' are hidden at disjoint subtrees of the compositional hierarchy.

[6] See also section 5.1 concerning action naming rules.

[7] In this case, T' is composed with the last node in the path upwards where the restriction is respected.

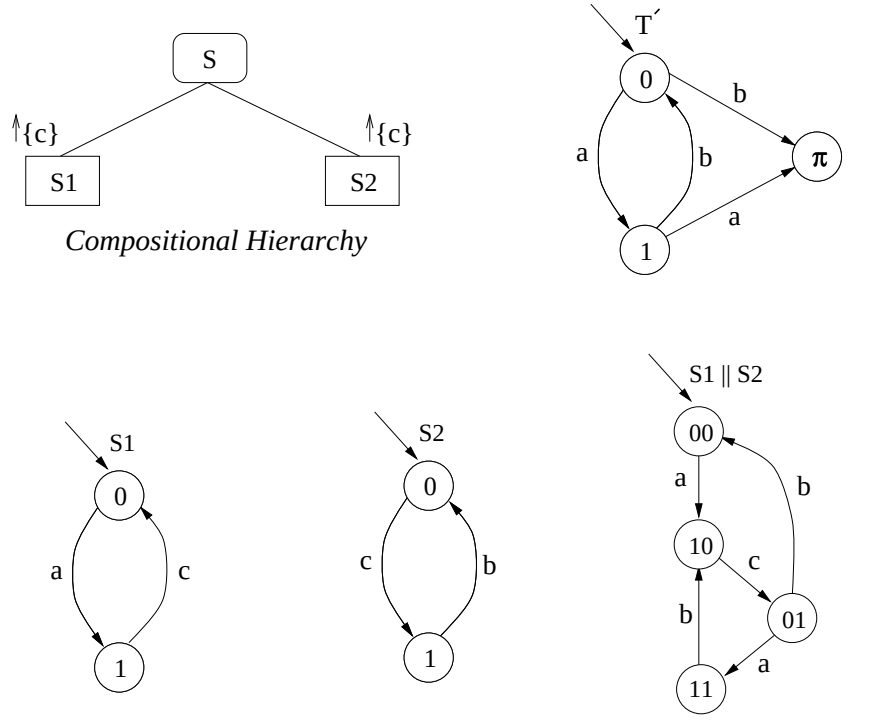

Figure 2: Different actions of T' are hidden at disjoint subtrees of the hierarchy. Trace (aca) of the system violates property T.

Such cases reveal the fact that the property cannot be satisfied by the system (see example in figure 2) and therefore there is no reason for proceeding with CRA before correcting the error.

As mentioned in section 5.1, we will always be able to express a system S in the form $Q \uparrow L$, where Q is a process in our model and L is a set of observable actions. Given that image process T' has been correctly located in the compositional hierarchy, the reachability graph obtained from CRA when T' has been included will be strongly equivalent to $(Q \parallel T') \uparrow L$. We can then prove that, similar to the case for interfaces in section 5.3, the technique described for locating violations of properties is not affected by the presence of the hiding operator.

As compared to verifying properties on the global graph obtained from CRA, our technique exhibits the following major advantage [6]: it allows for properties to be checked that concern actions which have been hidden at earlier stages of CRA. Moreover, it does not require any additional effort for checking properties after the computation of the global reachability graph. It therefore provides a uniform way of performing analysis.

5.5 Locating errors and violations

The techniques that have been described for detecting incorrect user-specified interfaces and violations of safety properties are related to the existence of the undefined state in the global reachability graph. Let us suppose that these techniques are used in the analysis of some system, where a number of properties are to be tested, and interfaces are introduced at various stages of the analysis. The presence of the undefined state in the final graph will reveal that at least one of the properties has been violated and/or at least one of the interfaces is incorrect. More specific information is needed, however, from an analysis method. More precisely:

- A transition to the undefined state is introduced by an incorrect interface. This reveals the fact that a legal trace of the system has been eliminated by the given interface process. In this case, we need to be able to identify the interface in question, as well as the exact transition that is missing from it, in order to correct it.

- A transition to the undefined state is introduced by the violation of some property. This reveals the fact that a legal trace of the system violates a property. In this case, we need to be able to identify the property in question, as well as the exact transition that leads the image process of the property to the undefined state.

We proceed in the same way for dealing with both of these cases [3, 6]. We locate errors and violations by keeping track of the relation between those transitions leading to the undefined state π in the reachability graph of any system or subsystem, and the transitions in the image processes[8] that have introduced them.

For convenience, we use $[P \xrightarrow{\alpha} \Pi]$ to denote the set of transitions in the image processes that contributed to the transition of $P \xrightarrow{\alpha} \Pi$. We call $[P \xrightarrow{\alpha} \Pi]$ *the set of ancestor transitions* of $P \xrightarrow{\alpha} \Pi$ [3]. In each of the CRA steps, the sets of ancestor transitions are updated according to the following rules:

Initialisation: For every image process $I' = \langle S, A, \Delta, q \rangle$ do the following: for every p, α such that $(p \xrightarrow{\alpha} \pi) \in \Delta$,

$$[P \xrightarrow{\alpha} \Pi] = \{(p \xrightarrow{\alpha} \pi)_{I'}\}$$

[8] These can correspond to either user-specified interfaces or safety properties.

(P is the process identified with state p in $\langle S, A, \Delta \rangle$). The subscript I' identifies the process that owns the transition.

Parallel composition: In the step of parallel composition, update

- $[P \parallel Q \xrightarrow{\alpha} \Pi]$ to $[P \xrightarrow{\alpha} \Pi]$ if $Q \not\xrightarrow{\alpha}$;
- $[P \parallel Q \xrightarrow{\alpha} \Pi]$ to $[Q \xrightarrow{\alpha} \Pi]$ if $P \not\xrightarrow{\alpha}$;
- $[P \parallel Q \xrightarrow{\alpha} \Pi]$ to $[P \xrightarrow{\alpha} \Pi] \cup [Q \xrightarrow{\alpha} \Pi]$ if $P \xrightarrow{\alpha} \Pi$ and $Q \xrightarrow{\alpha} \Pi$.

Hiding: In the step of hiding, update

- $[P \uparrow L \xrightarrow{\alpha} \Pi]$ to $[P \xrightarrow{\alpha} \Pi]$ if $\alpha \in L$;
- $[P \uparrow L \xrightarrow{\tau} \Pi]$ to $[P \xrightarrow{\alpha} \Pi]$ if $P \uparrow L \xrightarrow{\tau} \Pi$ is derived from $P \xrightarrow{\alpha} \Pi$ where $\alpha \notin L$.

Minimisation: In the step of process minimisation, update both $[P \xrightarrow{\alpha} \Pi]$ and $[Q \xrightarrow{\alpha} \Pi]$ to $[P \xrightarrow{\alpha} \Pi] \cup [Q \xrightarrow{\alpha} \Pi]$ if $P \sim Q$.

6 Conclusions and future work

We have presented the TRACTA technique for performing static analysis of concurrent systems. Processes are modelled as rooted labelled transition systems and synchronise/communicate in a *CSP*-like fashion. TRACTA efficiently addresses a number of the requirements for behaviour analysis techniques of concurrent systems, as illustrated in table 2.

Objectives	*We provide/propose:*
Simple and familiar computational model...	Labelled Transition Systems (LTS) • *close to state transition diagrams* • *simple graphical notation*
To facilitate reasoning about safety and liveness properties...	Reachability Analysis (easy detection of deadlocks) + *image automata* for checking safety properties
To support compositionality and incremental analysis...	Compositional Reachability Analysis (CRA)
To be supported by an efficient automated tool...	a way to reduce the intermediate state explosion problem: Contextual CRA which consists of • *automatically generating interfaces for subsystems* • *easily checking correctness of user-specified interfaces*
To provide multiple analysis capabilities...	an approximate non-expensive dataflow analysis technique for the error-prone early design stages [5, 2]

Table 2: Summary of the TRACTA technique.

The TRACTA technique has a sound underlying theory. Detection of incorrect interfaces and of violation of properties is performed in a uniform way, i.e. by composing image processes into the system. The largest part of the effort involved in detecting incorrect interfaces

and violations of properties is therefore incorporated in the process of computing a global reachability graph for the system.

A prototype has been implemented [2] that supports the largest part of the TRACTA technique. Our plans for the future are:

- To complete the existing prototype for full support of the TRACTA technique. Additionally, to implement a user-friendly graphical user interface for interaction with the user.
- To perform case studies that will show the performance of the TRACTA technique as compared to that of existing techniques. Case studies will also prove very useful in uncovering possible weaknesses of our modelling and/or analysis methods.

Future directions that we are considering include the following:

- To achieve a tight integration of exhaustive with approximate techniques [2, 5]. Useful information obtained at the early stages of analysis where approximate techniques are used could be fed to the stages of exhaustive analysis.
- To investigate the applicability of TRACTA to the feature interaction problem in telecommunications, as presented in [16].
- To provide a way of recovering lost details [14]. Reports of possible errors may be helpful only if they describe in detail how the error can occur. A common error reporting technique in analysis of concurrent systems is to present an example trace that exhibits an undesired property. Since suppression of detail may make such a trace less useful, we need to recover a detailed trace despite having suppressed details during the analysis. A method for recovering lost details has been proposed by Yeh in [14].
- To extend our technique for dealing with cases where processes communicate over imperfect channels (messages may be lost, delayed, ...). In such cases, properties may have to be verified only along those execution sequences that satisfy a special set requirements [7] (e.g. paths where a transmitted message is eventually received).

Acknowledgements

I gratefully acknowledge the guidance and advice of Prof. Jeff Kramer during the course of this work. Special thanks to Dr. Shing Chi Cheung for the useful discussions and helpful suggestions.

References

[1] G.R. Andrews. *Concurrent Programming - Principles and Practice.* The Benjamin/Cummings Publishing Company Ltd, 1991.

[2] S.C. Cheung. *Tractable and Compositional Techniques for Behaviour Analysis of Concurrent Systems.* PhD thesis, Dept. of Computing, Imperial College of Science, Technology and Medicine, 1994.

[3] S.C. Cheung and J. Kramer. Compositional Reachability Analysis of Finite-State Distributed Systems with User-Specified Constraints. Accepted for publication, 3d ACM International Symposium on the Foundations of Software Engineering, Washington, October 1995.

[4] S.C. Cheung and J. Kramer. Enhancing Compositional Reachability Analysis with Context Constraints. In *Proc. of the 1st ACM International Symposium on the Foundations of Software Engineering*, Los Angeles, California, December 1993.

[5] S.C. Cheung and J. Kramer. An Integrated Method for Effective Behaviour Analysis of Distributed Systems. In *Proc. of the 16th International Conference on Software Engineering (ICSE94)*, Sorrento, Italy, May 1994.

[6] S.C. Cheung and J. Kramer. Checking Sybsystem Safety Properties in Compositional Reachability Analysis. Submitted for publication, 1995.

[7] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[8] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional Model Checking. In *Proc. of the 4th Annual Symposium on Logic in Computer Science*, Pacific Grove, California, June 1989.

[9] S. Graf and B. Steffen. Compositional Minimization of Finite State Systems. In *Proc. of the 2nd International Conference of Computer-Aided Verification*, LNCS 531, pages 186–196, New Brunswick, NJ, USA, June 1990.

[10] C.A.R. Hoare. *Communicating Sequential Processes.* Addison-Wesley, 1985.

[11] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[12] J. Peterson. *Petri Net Theory and the Modelling of Systems.* Prentice-Hall, 1981.

[13] R.N. Taylor. Complexity of Analyzing the Synchronization Structure of Concurrent Programs. *Acta Informatica*, (19):57–84, 1983.

[14] W.J. Yeh. *Controlling State Explosion in Reachability Analysis.* PhD thesis, Purdue University, 1993.

[15] M. Young, R. Taylor, K. Forester, and D. Brodbeck. Integrated Concurrency Analysis in a Software Development Environment. In *Proc. ACM SISOFT'89, 3rd Symposium on Software Testing, Analysis and Verification (TAV3)*, pages 200–209, Florida, December 1989.

[16] P. Zave. Feature Interactions and Formal Specifications in Telecommunications. *IEEE Computer*, pages 20–28, August 1993.

A Theorems, rules and lemmas: full presentation

Theorem 5.1 *Let Z, I be two totally defined processes, where I is deterministic and free of internal action τ and $\sim$ denotes the strong equivalence relation. Then $Z \sim (Z \parallel I)$ iff:*

1. $\alpha I \subseteq \alpha Z$;
2. $tr(Z \upharpoonright \alpha I) \subseteq tr(I)$.

Proof:
if part: We will first prove that the two conditions are sufficient. To do so, we define a binary relation R over totally defined processes by $(Z, Z \parallel I) \in R$ iff:

(a) $\alpha I \subseteq \alpha Z$;

(b) $tr(Z \upharpoonright \alpha I) \subseteq tr(I)$;

(c) I is a deterministic process free of τs.

To show that R is a strong bisimulation relation, we will prove that it satisfies the three conditions of definition 4.12. Let $(Z, Z \parallel I) \in R$.

(1) $\alpha(Z \parallel I) \stackrel{\text{def}}{=} \alpha Z \cup \alpha I \stackrel{\alpha I \subseteq \alpha Z}{=} \alpha Z$.

(2) Does not apply here because all processes are totally defined.

(3) In order to show the satisfaction of condition *3(i)* we need to consider two cases:

Case 1: $Z \stackrel{\alpha}{\longrightarrow} Z'$ and $\alpha \notin \alpha I$.
Then $Z \parallel I \stackrel{\alpha}{\longrightarrow} Z' \parallel I$. Since $\alpha Z' \stackrel{\text{def}}{=} \alpha Z$ and $\alpha I \subseteq \alpha Z$, it is clear that $\alpha I \subseteq \alpha Z'$. Moreover, since $\alpha \notin \alpha I$, $tr(Z' \upharpoonright \alpha I) \subseteq tr(Z \upharpoonright \alpha I) \subseteq tr(I)$. Since I is deterministic and free of τ, we conclude that $(Z', Z' \parallel I) \in R$.

Case 2: $Z \stackrel{a}{\longrightarrow} Z'$ and $a \in \alpha I$.
It is clear that $a \neq \tau$ is a trace of Z, therefore since $tr(Z \upharpoonright \alpha I) \subseteq tr(I)$, a must also be a trace of I. I is deterministic and free of τ so $\exists I'$ such that $I \stackrel{a}{\longrightarrow} I'$ and $\forall I''$ such that $I \stackrel{a}{\longrightarrow} I''$, $I' = I''$. Therefore,

$$Z \parallel I \stackrel{a}{\longrightarrow} Z' \parallel I'.$$

$\alpha Z' \stackrel{\text{def}}{=} \alpha Z$ and $\alpha I' \stackrel{\text{def}}{=} \alpha I$ implies that $\alpha I' \subseteq \alpha Z'$. Since I is deterministic and free of τ, I' will obviously also be deterministic and free of τ.

It therefore remains to be shown that $tr(Z' \upharpoonright \alpha I') \subseteq tr(I')$ or equivalently that $tr(Z' \upharpoonright \alpha I) \subseteq tr(I')$. We will prove that by contradiction. Let us assume that $tr(Z' \upharpoonright \alpha I) \not\subseteq tr(I')$ which means that there exists a sequence of observable actions s such that $s \in tr(Z' \upharpoonright \alpha I)$ and $s \notin tr(I')$. But $a \in \alpha I$ so $a^\wedge s \in tr(Z \upharpoonright \alpha I)$. On the other hand, because I is deterministic, $s \notin tr(I')$ implies that $a^\wedge s \notin tr(I)$. Therefore $tr(Z \upharpoonright \alpha I) \not\subseteq tr(I)$, which contradicts the initial assumption that $(Z, Z \parallel I) \in R$. We conclude that $tr(Z' \upharpoonright \alpha I') \subseteq tr(I')$, and consequently $(Z', Z' \parallel I') \in R$.

Condition *3(ii)* can be proven in a similar way. $Z \parallel I \stackrel{\alpha}{\longrightarrow} Q$ implies one of the following:

- $\alpha \notin \alpha I$ and $Z \xrightarrow{\alpha} Z'$. Then $Q = Z' \parallel I$ and the proof proceeds as in case 1 above.
- $\alpha \in \alpha I$ which implies that $\alpha \in \alpha Z$ since $\alpha I \subseteq \alpha Z$. Then $Z \xrightarrow{\alpha} Z'$ and $I \xrightarrow{\alpha} I'$ therefore $Q = Z' \parallel I'$. The proof then proceeds as in case 2 above.
- $\alpha \notin \alpha Z$ and $I \xrightarrow{\alpha} I'$. Obviously, either $\alpha = \tau$ or $\alpha \in \alpha I$. Neither of the two can hold however because I is free of τ and $\alpha I \subseteq \alpha Z$. Therefore this case can never happen.

We have thus proven that for two totally defined processes Z, I where I is a deterministic process free of internal actions τ:

$$((\alpha I \subseteq \alpha Z) \wedge (tr(Z \uparrow \alpha I) \subseteq tr(I))) \implies Z \sim (Z \parallel I).$$

only if part: In this part we show that the two conditions are necessary. Let Z, I be two totally defined processes such that $Z \sim (Z \parallel I)$, where I is a deterministic process free of internal actions τ. We will prove that $\alpha I \subseteq \alpha Z$ and that $tr(Z \uparrow \alpha I) \subseteq tr(I)$.

(1) $Z \sim (Z \parallel I)$ implies that $\alpha Z = \alpha(Z \parallel I)$. So $\alpha Z = \alpha(Z \parallel I) = \alpha Z \cup \alpha I$ and therefore $\alpha I \subseteq \alpha Z$.

(2) We have shown in (1) that $\alpha I \subseteq \alpha Z$. By lemma A.3 we conclude that $tr(Z \uparrow \alpha I) \subseteq tr(I)$, which completes the proof.

We have tried to keep the conditions of the transparence theorem as weak as possible. We have managed it for all but the one that requires interface processes I to be deterministic and free of internal actions τ. This is the single condition that is not both necessary and sufficient. Figure 3 depicts an example where although I is non-deterministic, it is transparent to a process Z. On the other hand, figure 4 illustrates an example where although conditions 1 and 2 of the transparence theorem are satisfied, I is not transparent to Z because it is non-deterministic.

We conclude that it is not necessary for I to be non-deterministic and free of τ for it to be transparent to a process. However the two conditions of the transparence theorem cannot guarantee that I will be transparent if it is non-deterministic. This is the reason why we have decided to deal only with interfaces that are deterministic and free of internal actions τ.

We strongly believe that this is not a limitation of our technique . Interfaces, both user-specified and algorithmically derived, are introduced into the analysis in order to alleviate the intermediate state explosion problem. This is achieved by including in the analysis of subsystems the behaviour constraints imposed by its environment, in the form of interfaces. Given that τ actions never synchronise with other actions, they do not provide a way of reducing arbitrary interleaving of actions. Moreover introducing extra non-determinism into the system should be avoided. It is therefore reasonable to expect that interface processes should be deterministic and free of internal actions τ.

The transparence theorem is not always satisfied by processes that are not totally defined. This is shown by the example illustrated in figure 5. There is a limited number of cases in which undefined states need to be introduced into processes by our technique. Even though the transparence theorem holds for totally defined processes only, it has proven sufficient in dealing with all of these cases as has been shown in sections 5.3, 5.4. □

Theorem 5.2 *Let P, T be two totally defined processes, and T' be the image process of T, where $\alpha T \subseteq \alpha P$. Then $P \parallel T'$ does not have undefined traces iff $tr(P \uparrow \alpha T) \subseteq tr(T)$.*

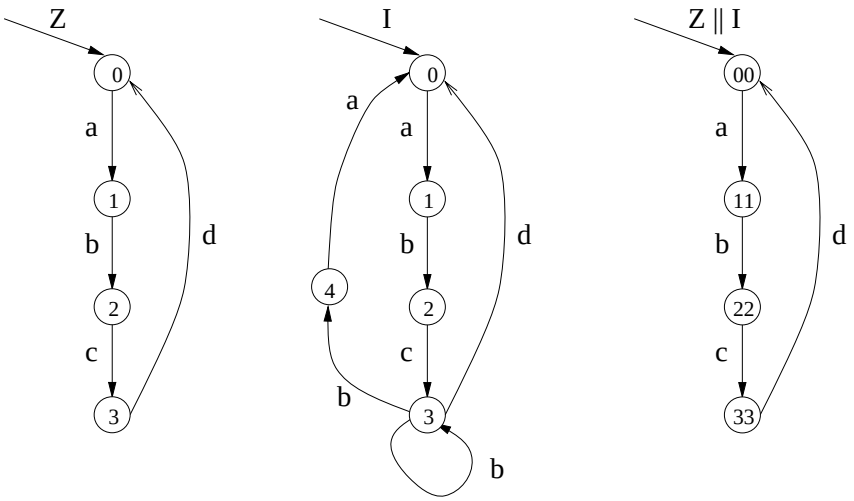

Figure 3: Transparent non-deterministic interfaces.

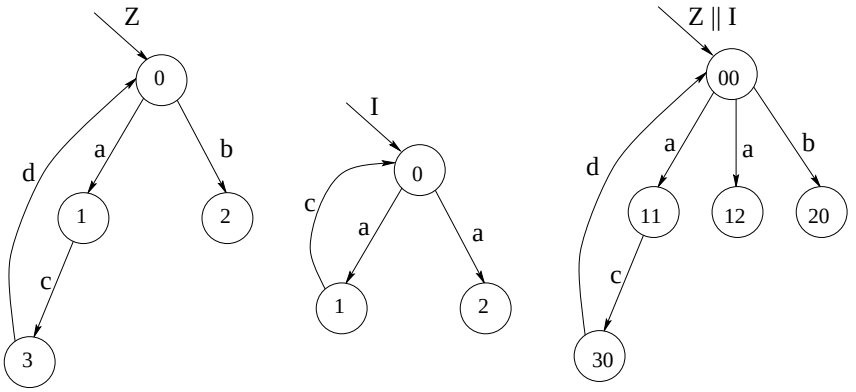

Figure 4: $\alpha I \subseteq \alpha Z$ and $tr(Z \uparrow \alpha I) \subseteq tr(I)$. However non-deterministic interface I is not transparent to Z.

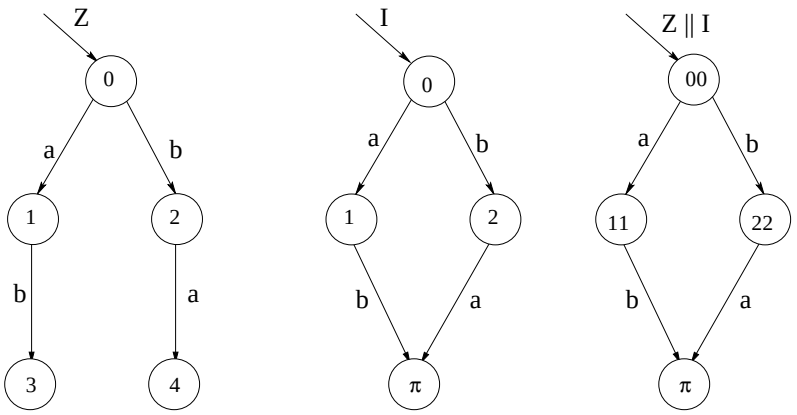

Figure 5: Processes that are not totally defined do not always satisfy the transparence theorem.

Proof:
if part: We assume that $tr(P \uparrow \alpha T) \subseteq tr(T)$. $\alpha T' = \alpha T \subseteq \alpha P$, so by lemma A.2

$$tr((P \parallel T') \uparrow \alpha T') = tr(P \uparrow \alpha T') \cap tr(T'). \tag{1}$$

By the supposition, $tr(P \uparrow \alpha T') \subseteq tr(T)$ and therefore

$$tr(P \uparrow \alpha T') \cap tr(T') \subseteq tr(T). \tag{2}$$

By equations 1, 2, we have:

$$tr((P \parallel T') \uparrow \alpha T') = tr(P \uparrow \alpha T') \cap tr(T') \subseteq tr(T). \tag{3}$$

P is totally defined, so a trace t in $(P \parallel T') \uparrow \alpha T'$ is undefined iff t is an undefined trace in T'. Due to the way in which T' is generated from T which is a totally defined process (definition 5.1, step 2), and because T' does not contain τ actions, t is an undefined trace in T' iff $t \in (tr(T') - tr(T))$. Therefore a trace t in $(P \parallel T') \uparrow \alpha T'$ is undefined iff $t \in (tr(T') - tr(T))$. But by equation 3, for any trace t in $(P \parallel T') \uparrow \alpha T'$, $t \in tr(T)$. So $(P \parallel T') \uparrow \alpha T'$ does not have undefined traces. By lemma 4.1, $(P \parallel T')$ does not have undefined traces either.

only if part: We assume that $(P \parallel T')$ does not have undefined traces. Suppose that $P \uparrow \alpha T$ can perform a trace that does not belong to $tr(T)$. Then obviously, by $\alpha T' = \alpha T$, $P \uparrow \alpha T'$ can also perform t. Let s be a prefix of t such that T can perform any proper prefix of s but not s itself. Due to the way in which T' is generated from T (definition 5.1, step 2), and since s is composed of actions that belong to αT, s is an undefined trace in T'. But then by the definition of the composition operator, s is also an undefined trace in $(P \uparrow \alpha T') \parallel T'$. By lemma A.1, $(P \uparrow \alpha T') \parallel T' = (P \parallel T') \uparrow \alpha T'$. As a result $(P \parallel T') \uparrow \alpha T'$ has undefined traces. Then by lemma 4.1 so does $(P \parallel T')$, which contradicts the assumption. We therefore conclude that $tr(P \uparrow \alpha T) \subseteq tr(T)$. □

Rule 5.1 *For any two processes P, Q and two sets of actions L, M:*

$P \uparrow L \parallel Q \uparrow M = (P \parallel Q) \uparrow (L \cup M)$ *iff* $((\alpha P - L) \cap (\alpha Q \cup M)) \cup ((\alpha Q - M) \cap (\alpha P \cup L)) = \emptyset$.

Proof:
only if part: Let us first assume that

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

Both sides of the equation compose the same processes P and Q, and differ on the stage at which actions are hidden. The left-hand side formula hides before composing, therefore we have to ensure that actions that are hidden on the left-hand side formula do not need to synchronise in the right-hand side one, i.e.,

$$(\alpha P - L) \cap \alpha Q = \emptyset \quad (4)$$
$$(\alpha Q - M) \cap \alpha P = \emptyset. \quad (5)$$

On the left-hand side equation, actions in $(\alpha P - L)$ and $\alpha Q - M$ are hidden for P, Q respectively. But on the right-hand side equation, actions in $\alpha P - (L \cup M)$ and $\alpha Q - (L \cup M)$ are hidden for P, Q respectively. Therefore, it must hold that

$$\alpha P - L = \alpha P - (L \cup M) \implies \alpha P - L = (\alpha P - L) - M \implies (\alpha P - L) \cap M = \emptyset \quad (6)$$
$$\alpha Q - M = \alpha Q - (L \cup M) \implies (\alpha Q - M) \cap L = \emptyset. \quad (7)$$

By equations 4, 5, 6 and 7, we conclude that

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

if part: Can be proven similarly. Moreover, it is trivial to show that the condition guarantees equality of alphabets for $(P \uparrow L \parallel Q \uparrow M)$ and $(P \parallel Q) \uparrow (L \cup M)$. □

Lemma A.1 *For any two processes P, A : $(P \parallel A) \uparrow \alpha A = (P \uparrow \alpha A) \parallel A$.*

Proof: By substituting αA for L, A for Q and αA for M in 5.1. The condition is obviously satisfied, therefore:

$$(P \uparrow \alpha A) \parallel (A \uparrow \alpha A) = (P \parallel A) \uparrow (\alpha A \cup \alpha A) \implies (P \uparrow \alpha A) \parallel A = (P \parallel A) \uparrow \alpha A.$$

□

Lemma A.2 *For any two processes P, T s.t. $\alpha T \subseteq \alpha P$, $tr((P \parallel T) \uparrow \alpha T) = tr(P \uparrow \alpha T) \cap tr(T)$.*

Proof: From lemma A.1, $tr((P \parallel T) \uparrow \alpha T) = tr((P \uparrow \alpha T) \parallel T)$.

$$tr((P \uparrow \alpha T) \parallel T) \stackrel{\text{def}}{=}$$
$$\{t \mid (t \uparrow \alpha(P \uparrow \alpha T) \in tr(P \uparrow \alpha T)) \wedge (t \uparrow \alpha T \in tr(T)) \wedge (t \in (\alpha(P \uparrow \alpha T) \cup \alpha T)^*)\} =$$
$$\{t \mid (t \uparrow (\alpha P \cap \alpha T) \in tr(P \uparrow \alpha T)) \wedge (t \uparrow \alpha T \in tr(T)) \wedge (t \in (\alpha T)^*)\} \stackrel{\alpha T \subseteq \alpha P}{=}$$
$$\{t \in (\alpha T)^* \mid (t \in tr(P \uparrow \alpha T)) \wedge (t \in tr(T)\} =$$
$$\{t \in (\alpha T)^* \mid t \in (tr(P \uparrow \alpha T) \cap tr(T))\}.$$

But $t \in (tr(P \uparrow \alpha T) \cap tr(T))$ implies that $t \in (\alpha T)^*$. We therefore conclude that:

$$tr((P \parallel T) \uparrow \alpha T) = tr((P \uparrow \alpha T) \parallel T) = tr(P \uparrow \alpha T) \cap tr(T). \qquad \square$$

Lemma A.3 *Let I, Z be two processes such that $\alpha I \subseteq \alpha Z$. Then:*

$$(Z \sim (Z \parallel I)) \Rightarrow (tr(Z \uparrow \alpha I) \subseteq tr(I)).$$

Proof: $(Z \sim (Z \parallel I)) \Rightarrow ((Z \uparrow \alpha I) \sim ((Z \parallel I) \uparrow \alpha I))$. But given that $\sim$ is stronger than trace equivalence,

$$tr(Z \uparrow \alpha I) = tr((Z \parallel I) \uparrow \alpha I). \tag{8}$$

Also, by the fact that $\alpha I \subseteq \alpha Z$ and by lemma A.2,

$$tr((Z \parallel I) \uparrow \alpha I) = tr(Z \uparrow \alpha I) \ \cap \ tr(I). \tag{9}$$

By equations 8 and 9, $tr(Z \uparrow \alpha I) = tr(Z \uparrow \alpha I) \ \cap \ tr(I) \Rightarrow tr(Z \uparrow \alpha I) \subseteq tr(I)$. $\square$