(To be published in the 16th IEEE International Conference on Software Engineering, Sorrento, Italy, May 1994)

An Integrated Method for Effective Behaviour Analysis of Distributed Systems

(Research Paper)

S.C. Cheung J. Kramer

Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, UK
Email: {scc, jk}@doc.ic.ac.uk

Abstract

Behaviour analysis is a valuable aid for the design and maintenance of well-behaved distributed systems. Dataflow and reachability analyses are two orthogonal but complementary behaviour analysis techniques. Individually, each of these techniques may be inadequate for the analysis of large-scale distributed systems. On the one hand, dataflow analysis algorithms, while tractable, may not be sufficiently accurate to provide meaningful detection of errors. On the other hand, reachability analysis, while providing exhaustive analysis, may be computationally too expensive for complex systems. In this paper, we present a method which integrates a dataflow and a reachability analysis technique to provide a flexible and effective means for analysing distributed systems at preliminary and final design stages respectively. We also describe some effective measures taken to improve the adequacy of the individual analysis techniques using concepts of action dependency and context constraints. A prototype supporting the method has been built and its performance is described in the paper. A realistic example of a distributed track control system is used as a case study.

1 Introduction

The design and construction of well-behaved concurrent and distributed systems is a difficult and error-prone process. We therefore believe that techniques and tools for analysing the behaviour of such systems are especially important. They can help to uncover logical flaws or unintended properties in a system's behaviour, such as deadlock, process starvation or synchronisation anomalies. In this paper, we focus on two orthogonal but complementary behaviour analysis techniques: reachability and dataflow analysis.

1.1 Reachability Analysis

The behaviour of a distributed system can be given in a *Labelled Transition System* (LTS for brevity). An LTS is a *state transition diagram* whose transitions are labelled by communicating actions. Reachability analysis is a popular technique for constructing a composite transition system from elementary components whose behaviours are given in LTSs [1, 13, 24, 32]. Traditionally, reachability analysis adopts an *all-at-once* approach where the global LTS (or reachability graph) of a system is constructed directly from the LTSs of its constituent processes in a single step. The global LTS describes all possible behaviours of the system as a composition of its constituents. Since modularity is not generally supported, the LTS given by the analysis usually contains too many states and transitions to be readily digested by software developers. Reachability analysis techniques also suffer from the combinatorial state explosion problem because the state space of a system can increase geometrically with the system size [31].

In view of these problems, modifications have been recently proposed to add compositionality and modularity to traditional reachability analysis techniques using various computational models in process algebra [25, 29, 34, 37]. These modifications enable a system to be decomposed into smaller subsystems and then successively composed and minimised. These modified techniques are known as *Compositional Reachability Analysis* (*CRA* for brevity). CRA allows developers to analyse and understand a system incrementally. For instance, developers may analyse and comprehend each subsystem individually, before inspecting the global behaviour of the enclosing (sub)system. Analysing a system in such a compositional way fits well with a constructive approach to software development [17].

However, the state explosion problem could be exacerbated as subsystems are locally minimised without considering their contexts or local environments [3, 9]. A locally minimised subsystem may contain a large number of behavioural traces forbidden by its context. In view of this problem, Sabnani et al. [29] suggested that a good composition order is essential in applying such a CRA technique. However, there is no systematic method or set of rules for the formulation of a good composition order such that state explosion problem can always be controlled. In this paper, we briefly present an effective method (section 5) to alleviate this problem by including context constraints in the composition of subsystems. These constraints can either be specified by users or automatically derived by a simple algorithm (section 5.4). Details of the method can be found in [2, 3].

1.2 Dataflow Analysis

Another alternative is to side-step the state explosion problem in reachability analysis by relaxing the accuracy of the analysis. Approximate techniques using dataflow analysis have been proposed [4, 21, 27, 28, 33]. These techniques approximate a system behaviour using simple polynomial dataflow algorithms. Some "unreachable" actions that are never executed by the system can be readily identified. However, the approximate analysis may either underestimate or overestimate the set of errors that may occur in the actual system. Despite this drawback, approximate dataflow analysis is useful for acquiring a preliminary understanding of system behaviour. It eliminates the need to apply expensive exhaustive analysis to those systems for which anomalies have been identified by the dataflow analysis. Among different dataflow analysis algorithms, the one proposed by Reif and Smolka [27] offers the best tractability. The complexity of the algorithm is linear to the size of the system. However, the accuracy of the algorithm is generally inadequate for distributed systems where messages are removed from the channels after reception. In this paper, we present a dataflow algorithm (section 4) using the principle of action dependency to alleviate this problem without significantly affecting its tractability. The complexity of the improved algorithm is quadratic to the size of the system. Preliminary and optimised versions of the dataflow analysis algorithm can be found in [4] and [2] respectively.

1.3 Integrated Approach

Different design stages may require different types of analysis support. For example, an approximate technique is not suitable for analysing behaviour of a distributed system at its final design stage because it does not deliver sufficient confidence in the correct behaviour of the design. We generally require an exhaustive technique. On the other hand, it is impractical to employ a costly exhaustive technique at the early stages of design as the specification at that stage is usually incomplete, tentative and plagued with errors. Indeed, it is unlikely that there is a single analysis technique fitting for all design stages. Thus, we need to use different analysis techniques at different stages of software design. This view is also held by Pezzè et al. [25] who suggest that a reasonable toolkit for analysis should provide several analysis capabilities, and that more conservative approximate approaches may be attempted before one resorts to an exhaustive state-space exploration such as reachability analysis. In this paper we present a method to integrate a dataflow and a reachability analysis technique to provide a flexible and effective means for analysing the design and implementation of distributed systems. The approximate technique is used at a preliminary stage when the design is unstable. The exhaustive technique is used at a final stage when the design becomes stable. Dataflow and reachability analyses were chosen to be the approximate and exhaustive techniques respectively in our experiment because they are complementary and share the same computational model. A distributed track control system, described in section 2, is used as a case study to illustrate and evaluate the method.

1.4 Paper Outline

In the next section, we describe briefly a state-based model of a distributed track control system. Section 3 presents an architecture in which dataflow and reachability analysis can be integrated. Section 4 gives an effective algorithm to detect unreachable actions using concepts of action dependency and history sets. The effectiveness of the algorithm for the analysis of preliminary designs is examined. Section 5 presents a technique to include context constraints in conventional CRA techniques. Performance of the technique in the exhaustive analysis of distributed systems is examined. Related work and conclusions are given in sections 6 and 7 respectively.

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 the states the process may reach and all the transitions it may perform. For instance, figure 1 represents an LTS of an engine which can be either in state 0 or 1. The engine can go from state 0 to 1 as the consequence of an action *start* and back to state 0 from 1 by an action *stop*.

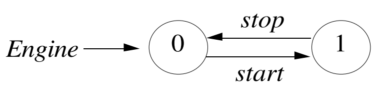

Figure 1: An LTS Description of an Engine

The set of actions which are considered relevant for a particular description of a process, say P, is called its *alphabet*, written as αP. 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.

Actions of a process can be restricted by a restriction operator $\uparrow$. $P \uparrow L$ is the process in which only the actions in set L are observable. Processes in a distributed system can also be composed by a composition operator $\|$ similar to that used in CSP [11]. $P \| Q$ 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. A formal definition of the LTS and the composition operator $\|$ is given in the appendix. In our discussion, we use the notation $\boldsymbol{tr}(P)$ to denote the set of traces exhibited by a process P. Similar computational models have been used in the literature [8, 9, 16, 26, 35] to specify and analyse parallel and distributed systems.

2.2 Behavioural Equivalences

Strong equivalence [22], denoted as ~, is commonly used to relate two processes whose behaviours are indistinguishable to an observer even if internal τ-actions are observable. It is generally accepted that strongly equivalent processes behave the same for all reasonable notions of observations [12, 26].

Observational equivalence [22], denoted as ≈, is commonly used to relate two processes whose behaviours are indistinguishable to an observer if internal τ-actions are not observable. Observational equivalence are widely used in CRA to abstract away internal actions during the local minimisation of subsystems. Observational equivalence is a weaker relation than the strong equivalence relation. Therefore, a pair of processes must be observational equivalent if they are strongly equivalent. That is,

$$P \sim Q \Rightarrow P \approx Q \ .$$

Further descriptions and illustrations of behavioural equivalences can be found in [8, 26, 30].

2.3 A Distributed Track control System

As an illustration of our discussion, we present a distributed track control system. The system was proposed by Fischer et al. [6] as a real-life case study for formal verification. Figure 2 shows the schematic of the distributed track control system. The system is modelled and specified in the same way as that in [6].

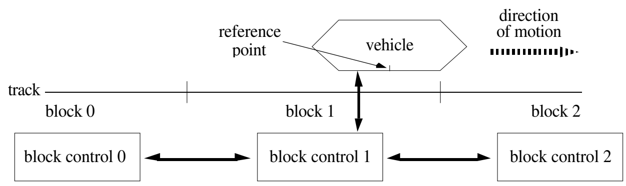

Figure 2: Communication between block controls and a vehicle

The track in the system is divided into sections, called *blocks*. The track system is jointly controlled by the blocks. Each block runs autonomously and coordinates its activities with its neighbours or vehicles through synchronised message passing. Although a vehicle may occupy two adjacent blocks at the same time, it can only communicate with the block in which the reference point of the vehicle is situated. For example, the vehicle in figure 2 occupies both block 1 and 2 but can only communicate with block 1. The transition of the reference point from one block to the next is assumed to be atomic. For convenience, we use α, β and δ to denote the *block number* of the predecessor, current and successor block of vehicle υ respectively in explaining the example. A vehicle υ exchanges five messages with the block β underneath as it travels along:

- υβ*_Enter* - The reference point of the vehicle enters the current block.
- υβ*_EnterComp* - The vehicle has completely entered the current block, the predecessor block is not occupied anymore. We assume that blocks are much longer than vehicles.
- υβ*_ResReq* - The vehicle requests the reservation of the successor block as it approaches the boundary of the current block.
- υβ*_OccSucc* - The vehicle has (partially) occupied the successor block. The distributed block control has to be designed in such a way that this action be performed only if the successor block is not occupied by another vehicle.
- υβ*_Exit* - The reference point of the vehicle exits the current block.

The two leading letters of a message (υ and β) indicate that vehicle υ is the sender and block β is the receiver. The vehicle υ also broadcasts a signal υ*_EinB*δ to an external observer indicating its position when it partially occupies the succeeding block δ. Figure 3 shows the vehicle's positions where the messages are transmitted.

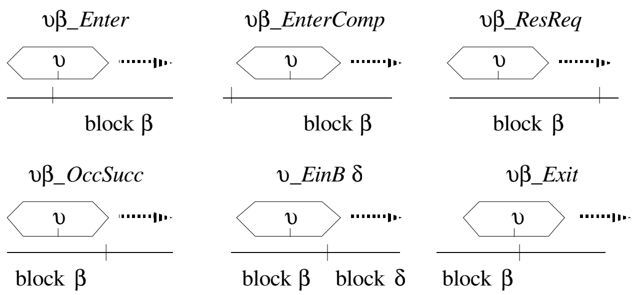

Figure 3: Messages transmitted by a vehicle υ

In our case study, the track contains a vehicle *V*, and is divided into three blocks: *Blk0*, *Blk1* and *Blk2*, which are connected as a ring. Figure 4 shows the interconnection diagram of the vehicles and the three blocks.

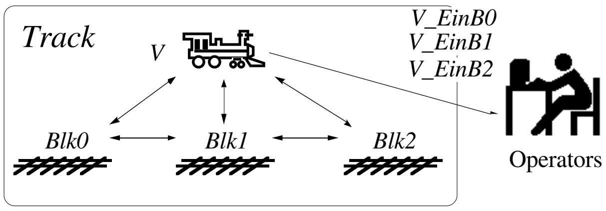

Figure 4: Interconnection diagram of the *Track* system with 1 vehicle and 3 blocks

The distributed track control system, *Track*, is defined as

$Track = (V \parallel Blk0 \parallel Blk1 \parallel Blk2\ ){\uparrow}ObvAct$
$ObvAct = \{V\_EinB0, V\_EinB1, V\_EinB2\ \}.$

Figure 5 shows the behaviour of a properly-behaved system on the circular track, assuming the vehicle *V* is initially placed on block 0.

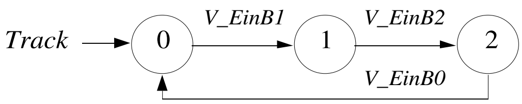

Figure 5: An LTS describing the global behaviour of *Track*

Figure 6 shows the behaviour of the vehicle *V* which starts travelling in block 0. Each of the three blocks comprises three components: *IntCtl*, *PreCtl*, *SucCtl*. Component *IntCtl* is responsible for internal control to ensure the main safety property that the current block never contains two or more vehicles at any time. *PreCtl* is responsible for predecessor control to monitor the vehicle position and releasing the predecessor block after the vehicle has completely entered into the current block. *SucCtl* is responsible for successor control to arrange the reservation of the successor block as the vehicle approaches the boundary of the current block. Behaviour of the three components is given in figure 7.

As before, the first and second letter of each message indicates its sender and receiver. For example, $\alpha\beta$_*ResReq* is a synchronised message from the predecessor block α to the current block β. If the predecessor and the current block is *Blk0* and *Blk1* respectively, then $\alpha\beta$_*ResReq* equals *01_ResReq*. Actions $\beta\beta$_*Reserve* and $\beta\beta$_*Release* are internal to the subsystem block β and hence not observable outside the scope of block β. To avoid collision of process names, processes *IntCtl*, *PreCtl* and *SucCtl* in subsystem block β are named *IntCtl*β, *PreCtl*β and *SucCtl*β respectively. The initial placement of vehicle *V* in block 0, as in [6], has been omitted to simulate a mistake in the preliminary design of the distributed track control system. In the following sections, we will describe how the behaviour of the track control system can be analysed using an integrated analysis technique.

3. Integrated Analysis

The concept of integrating approximate and exhaustive analyses is very promising but not well explored. In particular, we need different analysis capabilities at different stages of software design. In this paper, we investigate the potential of this approach using a dataflow technique for the preliminary, fast and cost-effective approximate analysis and a CRA technique for the more costly exhaustive analysis as shown in figure 8.

Actions that are never executed in the system are readily detected by a low-order polynomial dataflow analysis algorithm. Since the number of these unreachable actions must be less than the total number of distinct actions in a system, users are unlikely to be overwhelmed by a large number of reported anomalies as would be the case of exhaustive analysis.

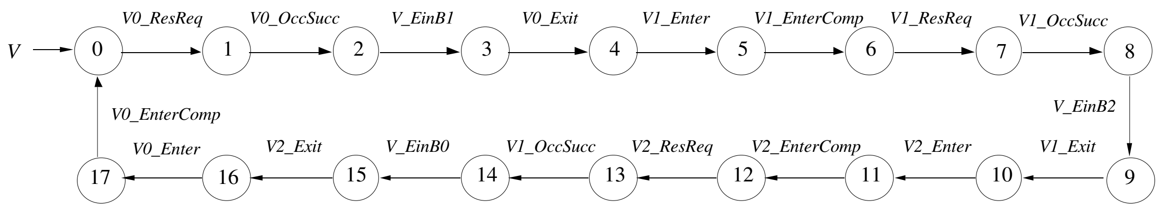

Figure 6: An LTS describing the behaviour of Vehicle *V*

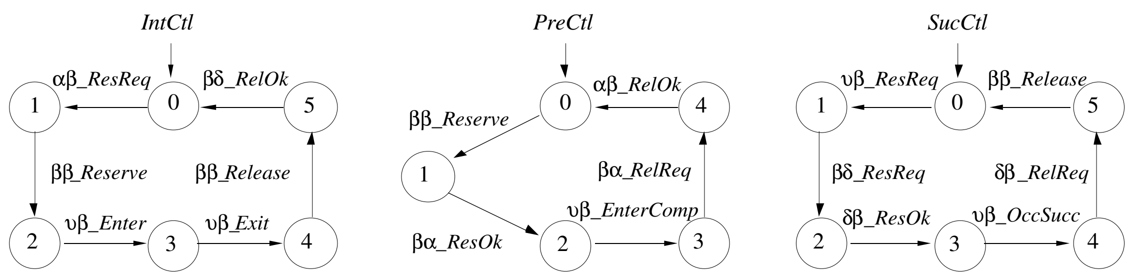

Figure 7: LTSs describing the behaviour of processes *IntCtl*, *PreCtl* and *SucCtl* in subsystem *Blk*β

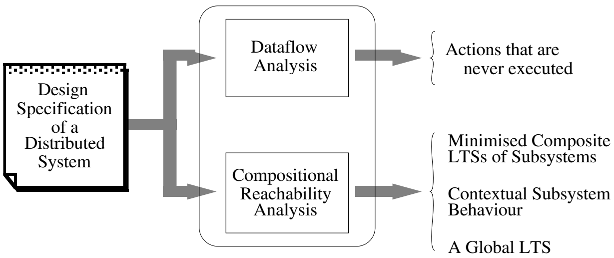

Figure 8: An Integrated Analysis Technique

When the system design becomes more stable, the users can conduct an exhaustive analysis using the CRA technique. Recent research has suggested that exhaustive analysis should be conducted incrementally starting from the elementary components at the leaves of the compositional structure [7, 8, 20, 34]. The analysis is conducted step by step in terms of subsystems such that behaviours of smaller subsystems are analysed before that of larger subsystems. A global LTS is generated at the final stage of the CRA.

4. Dataflow Analysis

Omission of the initial placement of the vehicle *V* in the distributed track control system (section 2) causes the system to deadlock with numerous *unreachable* actions. Actions are *unreachable* if they can never be executed in the system. Unreachable actions can be readily detected using an efficient dataflow analysis technique described in this section.

4.1 General Description

The dataflow analysis algorithm is general enough to handle systems with arbitrary loops or nested non-deterministic structures. Unreachable actions detected by the algorithm are conservative in the sense that these actions can never be executed by the actual system even when the data values of variables are considered. While detecting unreachable actions in a complexity quadratic to the system size, the algorithm may underestimate the set of unreachable actions. It refines an algorithm proposed by Reif & Smolka [27] to provide a better estimation of unreachable actions. The refinement is achieved by eliminating some spurious execution sequences in the dataflow analysis using the concepts of action dependency and history sets.

The action dependency relation records the order required between the first occurrence of an action and that of other actions. An action a is in the dependency set Δ_b if the action b can never be executed before a. There is a dependency set associated with each action in the system. An action cannot be executed unless all the actions in its dependency set have been executed. The history set, denoted as H_s, of a state s in an LTS T records all the actions that could have been executed by T before reaching state s. Actions not belonging to H_s can never be executed by T before reaching state s. Therefore, a transition $s{-}b{\rightarrow}s'$ in an LTS T can never occur if there is an action a in Δ_b such that (i) a does not belong to H_s and (ii) a belongs to the alphabet of T. Let us consider a simple system with two processes P and Q in figure 9. The history sets, initially empty, are shown at the top right corner of each state.

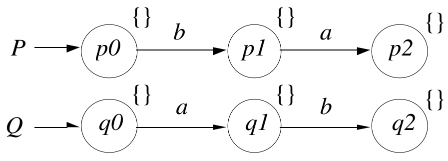

Figure 9: The LTSs of a simple system $P \parallel Q$

From the behaviour of Q, we can derive (using the *check* procedure in section 4.2) that $\Delta_b = \{a\}$. Similarly, we can derive from the behaviour of P that $\Delta_a = \{b\}$. The transition $p0{-}b{\rightarrow}p1$ cannot be executed because action a belongs to Δ_b and the alphabet of P but not to the history set of $p0$. Similarly, transition $q0{-}a{\rightarrow}q1$ cannot be executed either. As a result, no transition in the system can be executed. The history set of $p1$ would change from $\{\}$ to $\{b\}$ when the transition $p0{-}b{\rightarrow}p1$ has been executed in order to capture the information that P can reach state $p1$ after execution of action b.

4.2 Dataflow Algorithm

Let us denote Δ as the set containing Δ_a for all actions a in a system. Δ is an empty set when all Δ_as are empty. The *check* procedure (figure 10) identifies, according to the principles described in section 4.1, the unreachable actions in a given system Z subject to the supplied action dependency information Δ. When Δ is empty, the *check* algorithm works in the same way as that proposed by Reif and Smolka [27]. When Δ is non-empty, it offers a better estimation of unreachable actions using the additional information supplied by Δ. The worst case complexity of the algorithm is $O(|Act|+|States|+(|Act|\times|Trn|))$ where $|Act|$, $|Trn|$ and $|States|$ stand for the sum for distinct actions, transitions and states of component processes in a system respectively. The complexity is generally dominated by $O(|Act|\times|Trn|)$. If Δ is an empty set, the complexity becomes $O(|Act|+|States|+|Trn|)$.

Action dependency information can be derived using the *check* procedure based on the principle that action a must belong to a dependency set Δ_b if b is found to be unreachable when all transitions involving a are removed from the system. This principle is implemented by the outermost *for* loop in the procedure *main* (figure 10). After computing the dependency information, we can then invoke the *check* procedure again with a non-empty Δ.

procedure *check*(Z, Δ) /* Z is the input system and Δ is a set of Δ_a for all actions a in the system */
    mark every state s and action a in Z to be unreachable;
    mark every initial state s in Z to be reachable;
    set history set H_s of every state s to empty;
    repeat applying the following three rules until no further state or action can be marked reachable:

- An action a is *reachable* if there exists a transition $s\!-\!a\!\rightarrow\!s'$ in each component process P containing a in its alphabet such that s is reachable and $(\alpha P \cap \Delta_a) \subseteq H_s$.
- For any transition $s\!-\!a\!\rightarrow\!s'$, if both s and a are *reachable* and $(\alpha T \cap \Delta_a) \subseteq H_s$ then s' is *reachable*.
- For any transition $s\!-\!a\!\rightarrow\!s'$, if both s and a are *reachable* and $(\alpha T \cap \Delta_a) \subseteq H_s$ then $H_{s'}$ is assigned to $H_{s'} \cup H_s \cup \{a\}$

    end repeat
end *check*

procedure *main*(Z) /* Z is the system to be analysed */
    initialise all Δ_a to an empty set for each action a in Z;
    for each action a do /* work out the dependency set */
        prepare a new system Z' from the original system Z by removing all transitions involving a;
        run *check*(Z', Ø);
        for each unreachable action b identified in the previous step, *check*(Z', Ø), do
            insert a into Δ_b ;
    end for
    run *check*(Z, Δ);
end *main*

Figure 10: An Approximate Dataflow Analysis Algorithm

The memory required for the dataflow algorithm is dominated by the storage of history sets, which is $O(|States| \times |Act|)$. A prototype developed requires less than 100KBytes of memory to analyse a system of 128 processes each containing 64 states and an alphabet of 32 actions. Systems of that size are generally very difficult to analyse using exhaustive analysis techniques.

4.3 Approximate Analysis of Track Control System

Unreachable actions in the track control system can be readily identified by the dataflow analysis algorithm. Figure 11 shows the actions that are marked "reachable" at the termination of the algorithm.

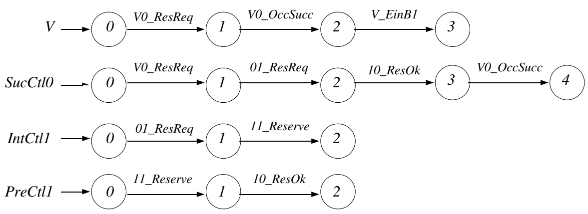

Figure 11: Possibly reachable actions in the Track Control system

Since the algorithm could underestimate the set of unreachable actions, actions that are marked reachable may or may not be actually executed in the original system. However, actions that are marked unreachable can never be executed in the original system. The traces in figure 11 are useful for understanding the underlying synchronisation error. The error is caused by an incorrect specification of the start state for the process *IntCtl0* in subsystem *Blk0*. *IntCtl0* should start at the state 3 rather than 0 (figure 7) because the block contains a vehicle initially. Figure 12 shows a correct behaviour of the process *IntCtl0*.

Figure 12: The correct behaviour of process *IntCtl0* in subsystem *Blk0*

4.4 Comparisons

The approximate dataflow algorithm takes 0.3 seconds to detect the design fault using a prototype running on a SUN Sparc/I workstation. For comparison, we have also analysed the system using conventional exhaustive CRA techniques [29, 37] by decomposing the system into the software hierarchy shown in figure 13.

In a conventional CRA [29, 37], subsystems are successively composed and minimised from the bottom components, such as *PreCtl0*, according to the software hierarchy. Actions internal to a subsystem are abstracted away from the composite LTS which is then minimised respecting observational equivalence. This minimised LTS can then be used to substitute the original subsystem in subsequent analysis. Such an analysis takes about 40 minutes to detect the design fault using the same workstation. The largest intermediate composite LTS contains 8,823 states and 40,579 transitions. A significant portion of the computational resources are consumed by the minimisation of the composite LTSs for subsystems *Blk01* and *Blk012*; both of them contain over 5,000 states and 20,000 transitions.

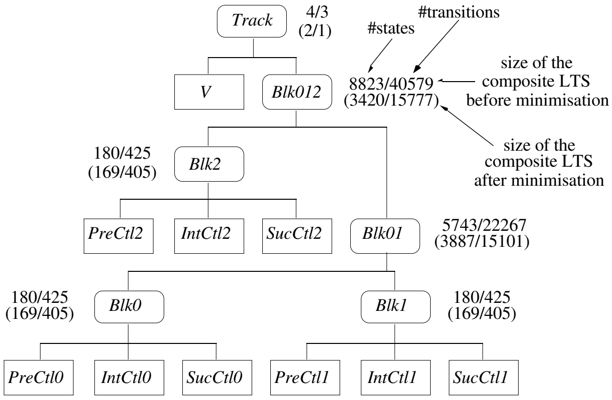

Figure 13: Software hierarchy of the faulty track control system for CRA

The computational time can be reduced significantly to around 5 seconds using an improved CRA technique described in the next section. Nevertheless, the time difference between the approximate dataflow analysis and exhaustive state space exploration technique increases rapidly as the track contains more blocks. This experiment thus demonstrates the advantages of analysing a preliminary design using approximate dataflow analysis before submitting the design for exhaustive analysis.

5 Contextual Compositional Reachability Analysis

After correcting the behaviour of *IntCtl0* in subsystem *Blk0* as shown in figure 12, no unreachable actions can be detected by the approximate dataflow algorithm. An exhaustive analysis technique, such as compositional reachability analysis (CRA), may now be employed to provide an extensive analysis of the system's properties, such as freedom from deadlock, freedom from starvation and freedom from undesirable executions. However as indicated in the previous section, the track control system cannot be efficiently analysed by conventional CRA techniques due to the generation of unnecessarily large intermediate transition systems.

5.1 Existing Problems with Conventional CRA Techniques

Each subsystem in a distributed program has to coordinate with other components. These components form a *context* of the subsystem and impose constraints on its behaviour. These constraints are known as *context constraints*. In conventional CRA, subsystems are locally minimised without considering their contexts. Subsystems thus obtained may contain a large number of behavioural traces which are forbidden by their contexts [3, 9]. Figure 14 shows the sizes of subsystems generated by the application of CRA to the track control system after correction. Comparing with figure 13, the correction changes the size of subsytem *Blk012* and the global system *Track*.

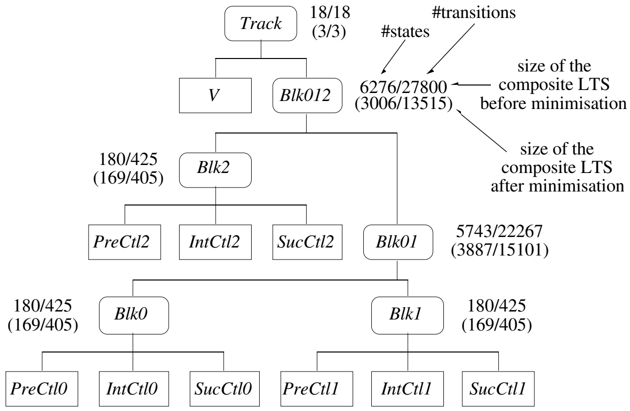

Figure 14: Sizes of composite subsystems generated by conventional CRA

In fact, many of the states and transitions in the subsystems generated by conventional CRA are forbidden by their contexts. For example, only 32 states and 53 transitions in the minimised composite LTS of *Blk0* would be preserved if the context of *Blk0* is considered during the construction of the composite LTS. The number of forbidden states and transitions is more significant in the subsystems *Blk01* and *Blk012*. In the minimised composite LTS of *Blk01*, only 66 states and 123 transitions are allowed by its context.

In the following, we propose an approach, called *contextual CRA*, to include context constraints in constructing the composite LTSs for intermediate subsystems. The approach is characterised by the following features:

- It performs the analysis using the same software hierarchy as that used by conventional CRA.
- It uses exactly the same computational model as that used by conventional CRA. Therefore software developers are not required to learn a more complex model in order to apply the contextual CRA technique.
- The global LTS constructed by contextual CRA is observationally equivalent to that constructed by a conventional CRA. Therefore, contextual CRA provides the same confidence as conventional CRA to the software developers in their designs.
- Context constraints are evaluated without an explicit construction of a composite LTS for the context. The evaluation is supported by a simple algorithm. Context constraints specified by software developers can be superimposed on those computed by the algorithm for a further elimination of forbidden subsystem behavioural traces.

5.2 Inclusion of Context Constraints

In a distributed program $S = P_1 \parallel ... \parallel P_n$, the construction of the composite LTS for a component, say P_1, can be regarded as a transformation of this component, say $\Psi(P_1)$. We wish to find a transformation process Ψ that can be used for eliminating forbidden traces in P_1 such that:

i. $\Psi(P_1)$ describes a more restricted behaviour than that of P_1;
ii. $\Psi(P_1)$ includes the context constraints of P_1;
iii. The overall system S behaves equivalently to the new system S' where $S' = \Psi(P_1) \parallel P_2 \parallel ... \parallel P_n$.

As a result, $\Psi(P_1)$ can be used as a substitute for P_1 within the context of S, and at the same time $\Psi(P_1)$ describes the behaviour of P_1 under the influence of its neighbours $P_2, ..., P_n$,

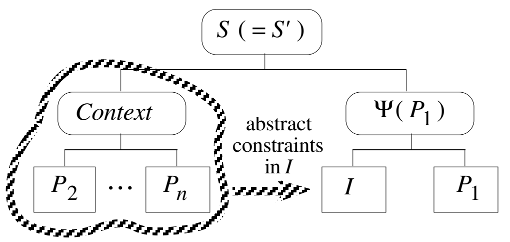

Figure 15: Inclusion of context constraints using a transparent interface process *I*

A natural choice for the transformation Ψ is the parallel composition operator $\parallel$ such that

$\Psi(P_1) = P_1 \parallel I$ and $\alpha I \subseteq \alpha P_1$

where I is a process capturing the context constraints of P_1 as shown in figure 15. Clearly this satisfies requirements (i) and (ii). In order that the introduction of I does not affect the behaviour of S (requirement iii), I must be *transparent* to S. Therefore, I is called a *transparent interface* process.

In figure 16, transparent interfaces introduced into the track control system are represented by shaded boxes. These interfaces can either be supplied by users or be derived algorithmically. In fact, transparent interfaces supplied by users and those algorithmically derived can be merged to form another interface imposing stricter constraints on the subsystem being minimised [2].

In the following, we will briefly present a technique to derive transparent interfaces and include them in subsystem composition. Comparing the number of states and transitions in figures 14 and 16, the technique presented significantly reduces the subsystem sizes in the distributed track control system.

5.3 An Interface Theorem

Theorem 1: Suppose S is a system and I is an interface process, and ~ denotes the strong behavioural equivalence,

$S \sim S \parallel I$

if

$\alpha I \subseteq \alpha S$;

$\boldsymbol{tr}(S \uparrow \alpha I) \subseteq \boldsymbol{tr}(I)$; *and*

I is a deterministic LTS free of internal action τ.

Theorem 1 gives three sufficient criteria for an interface I to be transparent to a system S. We use strong behavioural equivalence suggested by Milner [22]. The equivalence has been widely accepted as the strongest behavioural equivalence relation of interest to the theory of concurrency [26, 30]. The first criterion ensures that I does not introduce new actions into the S. The second criterion ensures that I does not eliminate any legal behavioural traces of S. The third criterion ensures that I does not introduce new non-determinism into S. Internal action τ is excluded from I because context constraints cannot be captured bytransitions labelled with τ. On the contrary, it may introduce divergence [11] into the system.

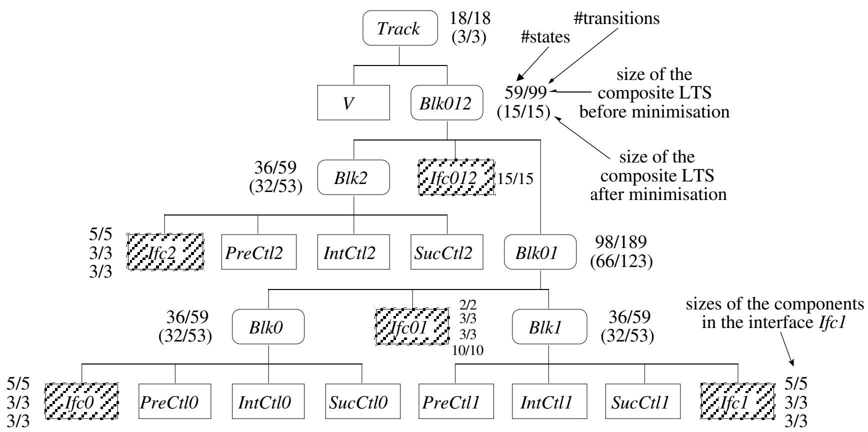

Figure 16: Software hierarchy of track control system with the introduction of transparent interfaces

If the third criterion is relaxed by allowing τ in I, the equivalence relation in the interface theorem becomes an observational equivalence [22]. Proofs and detailed descriptions of the theorem can be found in [2].

5.4 An Interface Construction Algorithm

Although an interface can be specified by users, it is more useful if the interface can be derived algorithmically because this allows the entire analysis task be automated. The derivation should avoid the need to construct an explicit composite LTS for the context in figure 15. This is because an LTS of the context may be non-deterministic and contain internal actions. Moreover this approach is not desirable as the computation for such a composite LTS generally suffers from the state explosion problem. Thus, we need to find a simple and practical algorithm to construct a transparent interface for a given context.

5.4.1 Algorithm Description

Suppose we want to construct an interface I for a contextual analysis of subsystem U in an application $W = U \parallel V$ where V is the context of U in W, the interface I can be constructed by the following two steps.

I Reducing Elementary Processes with respect to αU

Mark those elementary processes[1] of V which share two or more actions with αU in their LTSs. For each marked process, construct a reduced LTS by repeating the following two operations until no transition remains to be deleted:

(i) delete a transition $s_1 -a\rightarrow s_2$ where $a \notin \alpha U$;
(ii) merge states s_1 and s_2 together forming a single state.

As a result, each reduced LTS retains only those transitions with actions found in αU.

II Transforming Non-deterministic Reduced LTSs into Deterministic LTSs

Non-deterministic reduced LTSs obtained in step **I** can be transformed into deterministic LTSs using a standard NFA[2] to DFA[3] algorithm [14]. Each deterministic LTS can then be minimised by a DFA minimisation algorithm [14].

The interface I is then given by a parallel composition of the minimised deterministic LTSs obtained in step **II**. Note that I satisfies (i) $\alpha I \subseteq \alpha V$, (ii) $\boldsymbol{tr}(V\!\uparrow\!\alpha I) \subseteq \boldsymbol{tr}(I)$, and (iii) *$I$ is a deterministic LTS free of internal action τ.* Hence I is transparent to V by theorem 1, i.e. $V \sim V \parallel I$. This implies I is also transparent to W, i.e. $W \sim W \parallel I$. The construction algorithm does not require a derivation of an explicit LTS for V in constructing the interface I.

Only those processes sharing more than one action with the target subsystem U are chosen because a reduced LTS can only restrict the behaviour of U if it contains two or more states. A process sharing one action or none with U is likely to be reduced into an LTS with a single state. Such a choice does not affect the correctness of the algorithm but it saves the computational effort by skipping those processes which are unlikely to impose the behavioural constraints onto the target subsystem U. The proof of correctness of this algorithm can be found in [2].

5.4.2 Algorithm Complexity

The complexity of step **I** is at most linear with respect to the size of V. The minimisation of each deterministic LTS in step **II** can be computed in $O(N\log N)$ where N is the number of states in the LTS. Therefore the algorithmic complexity is dominated by the NFA to DFA transformation in step **II**. The theoretical complexity of the transformation is exponential to the number of states in a non-deterministic reduced LTS. In practice, the computational costs for the transformation are low because the reduced LTS is often (i) nearly deterministic and (ii) small in size, unless the elementary processes to be reduced shares many actions with the target subsystem U and contain many states and transitions. This algorithm therefore provides an effective and pragmatic way to construct interfaces from those contexts composed of small or medium size elementary processes.

5.4.3 Example - Derivation of interface *Ifc0* for subsystem analysis of *Blk0*

Among the 10 elementary processes (white square boxes in figure 16) in the track control system, only processes V, *PreCtl1*, and *SucCtl2* share two or more actions with *Blk0*. Applying the interface construction algorithm to V (figure 6), state 2 is merged with 3 to form a single state because *V_EinB1* is not an action in the alphabet of *Blk0*. Similarly, states 4 to 16 are merged together to form another single state. As a result, a reduced LTS V' in figure 17 is constructed. Similarly, the algorithm derives the reduced LTSs *PreCtl1'* and *SucCtl2'* based on *PreCtl1* and *SucCtl2* respectively in figure 7. The interface *Ifc0* is therefore equal to ($V' \parallel$ *PreCtl1'* $\parallel$ *SucCtl2'*) and therefore the composite LTS for *Blk0* is evaluated as a composition of (*IntCtl0* $\parallel$ *PreCtl0* $\parallel$ *SucCtl0* $\parallel V' \parallel$ *PreCtl1'* $\parallel$ *SucCtl2'*).

1 Elementary processes are those processes at the leaves of an hierarchy.
2 NFA stands for Non-deterministic Finite Automata.
3 DFA stands for Deterministic Finite Automata.

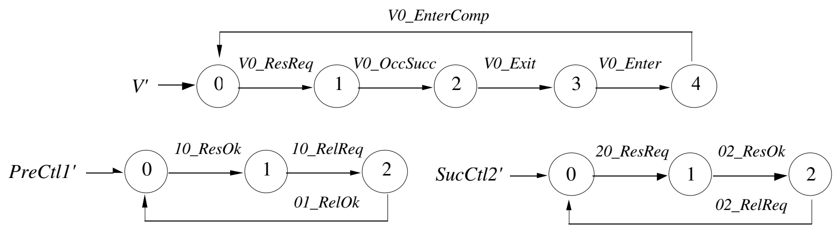

Figure 17: Reduced LTSs derived for constructing the composite LTS of *Blk0*

	Blk0, *Blk1*, *Blk2*	*Blk01*	*Blk012*	*Track*
Conventional CRA (figure 14)	180 / 425	5743 / 22267	6276 / 27800	18 / 18
Contextual CRA (figure 16)	36 / 59	98 / 189	59 / 99	18 / 18

Figure 18: Subsystems' sizes generated by conventional & contextual CRA techniques

5.5 Exhaustive Analysis of Distributed Track Control System

Figure 16 gives the sizes of the intermediate subsystems generated using contextual CRA technique. It takes 5 seconds to conduct a complete analysis of the distributed track control system for a prototype running on a SUN Sparc/I workstation. The 5 second elapsed time includes the time for context constraint evaluation. At the end of the analysis, the global LTS in figure 5 is obtained. Similar analysis using conventional CRA technique takes about 25 minutes and the sizes of the intermediate subsystems are given in figure 14. The table in figure 18 compares the sizes (#states/#transitions) of intermediate subsystems generated by conventional and contextual CRA techniques in the analysis of the distributed track control system. The effectiveness of the contextual CRA technique is also confirmed by a case study of a clients/server system [3].

6 Related Work

Dataflow algorithms have been proposed for the detection of deadlocks and infeasible execution sequences in a system [15, 23, 36]. However, the complexity of these algorithms are generally exponential with the size of a system. Taylor [32] suggests that exhaustive analysis of general distributed systems by dataflow algorithms is very difficult. Reif et al. [27] put forward a dataflow algorithm for approximate analysis of a system. The algorithm is tractable with linear complexity with system size.

The accuracy of Reif's algorithm in detecting unreachable actions is improved with concepts of action dependency and history sets. The improved algorithm demonstrates a quadratic complexity with system size. A preliminary version of the algorithm which requires a unique label for each pair of communicating actions, can be found in [4]. This requirement is eliminated in the dataflow algorithm presented in this paper.

Compositional or related analysis techniques have been proposed in [29, 37]. However, context constraints are not considered in local subsystem minimisation. As a result, the LTS of a subsystem may be even larger than that of the whole system.

Larsen et al. [19] have presented a method to parameterise the bisimulation relations using an user-given interface process through which the context constraints are specified. The minimised LTS for a subsystem is also supplied by users.

Clarke et al. [5] presented an approach to verify certain global system properties in local subsystem analysis where the context constraints are considered. The generation of an interface process assumes the existence of an LTS for the environment. There is no suggestion as to how to generate a minimised LTS for a subsystem so as to simplify subsequent analysis.

Graf et al. [9] proposed an approach to assist compositional analysis using context constraints. The interface processes expressing those context constraints have to be supplied by users. This poses a problem if one wishes to automate the analysis.

To avoid this problem, we therefore propose a simple algorithm to evaluate the context constraints in the contextual CRA. A more detailed description of the algorithm together with a simple clients/server example and preliminary performances can be found in [3]. In this paper, we have demonstrated the contextual CRA technique using a more complex real-life example and studied the integration with an enhanced, approximate dataflow technique for the analysis of large-scale distributed systems.

The integrated method is an attempt to combine two orthogonal but complementary analysis paradigms: approximate and exhaustive analysis for verification and validation of distributed systems. The approximate analysis is realised by an improved dataflow analysis technique. The exhaustive analysis is realised by a contextual CRA technique. So far, we have found no literature describing similar integrated approaches.

7. Conclusion

This paper presents a flexible and effective behaviour analysis method to support the design process of large-scale distributed systems. The method integrates two techniques: approximate dataflow and exhaustive reachability analysis. The adequacy of these techniques have been improved using concepts of action dependency and context constraints respectively.

Dataflow analysis provides a fast preliminary analysis of the global system. The analysis reveals certain unreachable actions and deadlock scenarios. The dataflow algorithm detects unreachable actions with a complexity quadratic to the system size. The algorithm is useful for identifying less sophisticated errors in a preliminary design. This is confirmed by an experiment to identify synchronisation errors in a preliminary specification of a distributed track control system. It takes 0.5 seconds for the dataflow algorithm to identify these errors as compared to 40 minutes taken by a conventional CRA technique or 5 seconds taken by a contextual CRA technique. The time difference between the approximate dataflow analysis and the exhaustive CRA techniques increases rapidly with the increase of the number of blocks in the track system.

Compositional reachability analysis (CRA) advocates incremental analysis of a system in a compositional manner. The analysis fits well with a constructive approach to software development. Contextual CRA improves the conventional CRA by including context constraints in constructing the composite LTSs for intermediate subsystem. The contextual CRA technique can significantly reduce the computational efforts required for exhaustive analysis. This is confirmed by a case of the distributed track control systems.

We strongly believe that an integration of the proposed dataflow algorithm and contextual CRA techniques can provide a more flexible and effective analysis support for the construction of large-scale distributed systems. We are interested in applying the techniques to Statecharts [10], a design formalism using similar communication semantics and graphical notations. To further explore the potential of the integrated analysis method, we are also applying the technique 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* [18].

Acknowledgement

We would like to thank the anonymous referees for their suggestions to improve the quality of the paper. We would also like to acknowledge the Croucher Foundation, Swiss Bank and DTI (Grant Ref: IED 410/36/2) for their financial support.

References

[1] K.R. Apt, "A Static Analysis of CSP Programs," in *Workshop Program Logic.*, Outtsvyrgg, June 1983.

[2] S.C. Cheung, "Tractable and Compositional Techniques for Behaviour Analysis of Concurrent Systems," Ph.D. thesis, Department of Computing, Imperial College of Science, Technology and Medicine, 1994.

[3] S.C. Cheung and J. Kramer, "Enhancing Compositional Reachability Analysis with Context Constraints," in *the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM SIGSOFT, Los Angeles, California, December 1993, pp. 115-125.

[4] S.C. Cheung and J. Kramer, "Tractable Flow Analysis for Anomaly Detection in Distributed Programs," in *4th European Software Engineering Conference (ESEC'93)*, Germany, September 1993, LNCS 717, Springer-Verlag, pp. 283-300.

[5] E.M. Clarke, D.E. Long and K.L. McMillan, "Compositional Model Checking," in *4th Annual Symposium on Logic in Computer Science.*, IEEE, Pacific Grove, California, June 1989, pp. 353-362.

[6] S. Fischer, A. Scholz and D. Taubner, "Verification in Process Algebra of the Distributed Control of Track Vehicles - A Case Study," in *4th International Workshop on Computer Aided Verification*, Montreal, Canada, June/July 1992, pp. 192-205, published in LNCS 663.

[7] L. Fix, N. Francez and O. Grumberg, "Program Composition and Modular Verification," in *18th International Colloquium on Automata, Languages and Programming.*, Madrid, Spain, July 1991, pp. 93-114, published in LNCS 510.

[8] C. Ghezzi, M. Jazayeri and D. Mandrioli, *Fundamentals of Software Engineering, Chapter 6*, Prentice-Hall Inc., 1991.

[9] S. Graf and B. Steffen, "Compositional Minimization of Finite State Systems," in *2nd International Conference, CAV*, New Brunswick, NJ, USA, June 1990, pp. 186-196, published in LNCS 531.

[10] D. Harel, "On Visual Formalisms," *Communication of the ACM*, vol. 31, no. 5, 514-530, 1988.

[11] C.A.R. Hoare, *Communication Sequential Processes*, Prentice-Hall, 1985.

[12] S. Holmstöm, "Hennessy-Milner Logic with Recursion as a Specification Language, and a Refinement Calculus Based on It," in *BCS-FACS Workshop on Specification and Verification of Concurrent Systems*, Stirling, Scotland, July 1988, pp. 294-330.

[13] G. Holzmann, *Design and Validation of Computer Protocols*, vol. Chapter 8 and 11, Prentice Hall, 1991.

[14] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 1979.

[15] P.C. Kanellakis and S.A. Smolka, "On the Analysis of Cooperation and Antagonism in Networks of Communicating Processes," in *4th ACM Symposium on Principles of Distributed Computing*, Ontario, Canada, August 1985, pp. 23-38.

[16] J. Kemppainen, M. Levanto, A. Valmari, et al., ""ARA" Puts Advanced Reachability Analysis Techniques Together," in *5th Nordic Workshop on Programming*

Environment Research., Programming Environment Research, Tampere University of Technology, Tampere, Finland, January 1992, pp. 233-257.

[17] J. Kramer and J. Magee, "A Constructive Approach to the Design of Distributed Systems," in *10th IEEE International Conference on Distributed Computing Systems.*, Paris, June 1990.

[18] J. Kramer, J. Magee, K. Ng, et al., "The System Architect's Assistant for Design and Construction of Distributed Systems," in 4th IEEE Workshop on Future Trends of Distributed Computing Systems, Lisbon, September 1993.

[19] K. Larsen and R. Milner, "Verifying a Protocol Using Relativized Bisimulation," in *14th International Colloquium on Automata, Languages and Programming*, Karlsruhe, Federal Republic of Germany, July 1987, pp. 126-135, published in LNCS 267.

[20] K.G. Larsen, "Compositional Theories based on an Operational Semantics of Contexts," in *REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Mook, The Netherlands, May/June 1989, pp. 487-518, published in LNCS 430.

[21] S.P. Masticola and B.G. Ryder, "A Model of Ada Programs for Static Deadlock Detection in Polynomial Time," in *ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, California, May 20-21, 1991.

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

[23] W. Peng and S. Purushothaman, "Data Flow Analysis of Communicating Finite State Machines," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 3, 399-432, 1991.

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

[25] M. Pezzè, R.N. Taylor and M. Young, "Graph Models for Reachability Analysis of Concurrent Programs," Tech. Rep., Software Engineering Research Centre, Department of Computer Science, Purdue University, May 1993.

[26] A. Rabinovich, "Checking Equivalences Between Concurrent Systems of Finite Agents," in *19th International Colloquium on Automata, Languages and Programming*, Wien, Austria, July 1992, pp. 696-707, published in LNCS 623.

[27] J. Reif and S. Smolka, "Data Flow Analysis of Distributed Communicating Processes," *International Journal of Parallel Programming*, vol. 19, no. 1, 1-30, 1990.

[28] J.H. Reif, "Data Flow Analysis of Communicating Processes," in *6th ACM Symposium on Principles of Programming Languages*, June 1979, pp. 257-268.

[29] K.K. Sabnani, A.M. Lapone and M.Ü. Uyar, "An Algorithmic Procedure for Checking Safety Properties of Protocols," *IEEE Transactions on Communications*, vol. 37, no. 9, 940-948, September 1989.

[30] D. Taubner, *Finite Representation of CCS and TCSP Programs by Automata and Petri Nets*, LNCS 369, Springer-Verlag, 1989.

[31] R.N. Taylor, "Complexity of Analyzing the Synchronization Structure of Concurrent Programs," *Acta Informatica*, vol. 19, 57-84, 1983.

[32] R.N. Taylor, "A General-Purpose Algorithm for Analyzing Concurrent Programs," *CACM*, vol. 26, 362-376, 1983.

[33] R.N. Taylor and L.J. Osterweil, "Anomaly Detection in Concurrent Software by Static Data Flow Analysis," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 3, 265-278, 1980.

[34] A. Valmari, "Compositional State Space Generation," Tech. Rep., Department of Computer Science, University of Helsinki, Finland, A-1991-5, October 1991.

[35] A. Valmari, "Alleviating State Explosion during Verification of Behavioural Equivalence," Tech. Rep., Department of Computer Science, University of Helsinki, Finland, A-1992, August 1992.

[36] R.D. Yang and C.G. Chung, "The Analysis of Infeasible Concurrent Paths of Concurrent Ada Programs," in *14th Annual International Computer Software and Applications Conference (COMPSAC 90)*, Chicago, Illinois, October 1990, pp. 424-429.

[37] W.J. Yeh and M. Young, "Compositional Reachability Analysis Using Process Algebra," in *Symposium on Testing, Analysis, and Verification (TAV4)*, ACM SIGSOFT, Victoria, British Columbia, October 8-10, 1991, pp. 49-59.

Appendix - Semantics of Parallel Composition

Formally, a *labelled transition system* (LTS) T of a process P is a quadruple $< S, A \cup \{\tau\}, \rightarrow, P >$ where

(i) S is a set of states;

(ii) A is a communicating alphabet of P, which does not contain the internal action τ;

(iii) $\rightarrow \subseteq S \times A \cup \{\tau\} \times S$ is a mapping associating with each $a \in A \cup \{\tau\}$;

(iv) P is the initial state of the labelled transition system T.

The parallel composition $T_1 \parallel T_2$ of two LTSs $T_1 = (S_1, A_1, \rightarrow_1, z_1)$ and $T_2 = (S_2, A_2, \rightarrow_2, z_2)$ is defined as $(S, A, \rightarrow, z)$ where

- $S = S_1 \times S_2$
- $A = A_1 \cup A_2$
- $z = (z_1, z_2)$
- $\rightarrow$ is given by the following three transition rules:

$$\frac{P -a\rightarrow_1 P'}{P \parallel Q -a\rightarrow P' \parallel Q} \; (a \notin A_2)$$

$$\frac{Q -a\rightarrow_2 Q'}{P \parallel Q -a\rightarrow P \parallel Q'} \; (a \notin A_1)$$

$$\frac{P -a\rightarrow_1 P' \quad Q -a\rightarrow_2 Q'}{P \parallel Q -a\rightarrow P' \parallel Q'} \; (a \in A_1 \cap A_2)$$