

# Behaviour Analysis of Distributed Systems using the Tracta Approach

Dimitra Giannakopoulou, Jeff Kramer

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

Email: {dg1, jk}@doc.ic.ac.uk

Shing Chi Cheung

Department of Computer Science  
Hong Kong University of Science and Technology,  
Clear Water Bay, Hong Kong

Email: scc@cs.ust.hk

## Abstract

*Behaviour analysis should form an integral part of the software development process. This is particularly important in the design of concurrent and distributed systems, where complex interactions can cause unexpected and undesired system behaviour. We advocate the use of a compositional approach to analysis. The software architecture of a distributed program is represented by a hierarchical composition of subsystems, with interacting processes at the leaves of the hierarchy. Compositional reachability analysis (CRA) exploits the compositional hierarchy for incrementally constructing the overall behaviour of the system from that of its subsystems. In the Tracta CRA approach, both processes and properties reflecting system specifications are modelled as state machines. Property state machines are composed into the system and violations are detected on the global reachability graph obtained. The property checking mechanism has been specifically designed to deal with compositional techniques. Tracta is supported by an automated tool compatible with our environment for the development of distributed applications.*

## 1. Introduction

### Background

Distributed processing is widely used to provide computing support for diverse applications. Many of these applications are complex and critical; an error can have catastrophic consequences. Behaviour analysis is a useful technique that can help discover defects and check if a system performs as intended. Such analysis can be applied to a finite state model of the system.

Reachability analysis consists of constructing the product state space of all components in the system, where the behaviour of each component is modelled as a finite state interacting process. Automated tools provide significant help in making the analysis of large systems manageable. Reachability analysis owes much of its popularity to the fact that it is fairly easy to automate.

The main disadvantage of this technique is state explosion, the exponential relation between the system state space and the number of its constituent components. Great effort has been made to avoid this problem by not having to construct the complete state graph. Automated techniques that reduce the size of the graph to be explored without compromising exhaustive search can be roughly classified into two approaches. *Reduction by partial orders* avoids the generation of all paths formed by the interleaving of the same set of transitions (Godefroid and Wolper 91, Holzmann, et al. 92). *Reduction by compositional minimisation* bases reduction on intermediate simplification of subsystems (Cheung and Kramer 96b, Graf and Steffen 90, Krimm and Mounier 97, Valmari 93b). Techniques in the latter category are known as *compositional reachability analysis* (CRA). CRA is a good complement to a compositional software development process.

In CRA, the system is decomposed into a hierarchy of subsystems. The behaviour of the system is then composed stepwise from those of its subsystems in a bottom-up manner. At every intermediate stage of the analysis, internal

details of subsystem behaviour are hidden and the subsystem is minimised. The observable behaviour of a subsystem where internal details are hidden can generally be represented by a smaller state machine. The key to the success of the technique is therefore to hide as many internal actions as possible in each subsystem.

Although CRA may significantly reduce the state space of the global system, it is susceptible to intermediate state explosion. Intermediate state explosion occurs when components of the system explode faster than the system itself. When constrained by activities of their context, these components usually have a much smaller state space. Various techniques have been proposed for addressing this problem (Cheung and Kramer 96b, Graf and Steffen 90, Krimm and Mounier 97, Yeh 93a). The most successful rely on composing intermediate subsystems with processes that restrict their behaviour according to their context. Such processes, which we refer to as contextual interfaces, can be automatically generated (Cheung and Kramer 96b, Krimm and Mounier 97) or user-specified (Cheung and Kramer 96b, Graf and Steffen 90, Krimm and Mounier 97).

Reachability analysis and CRA concentrate on the construction of a finite model for a system, based on the models of its components. In order to analyse the finite model of a system for desired properties, two general approaches are used in practice (Clarke and Wing 96a). In the first approach, properties are expressed in a temporal logic, and an efficient search procedure is used to check if a given state transition system is a model for the specification. In the second approach, the specifications are also given as state transition systems. In this case, the system is compared to the specification to determine whether its behaviour conforms to the behaviour of the specification. Vardi and Wolper in (Vardi and Wolper 86) establish an automata theoretic approach to verification, which consists of expressing temporal logic with automata, and using these to analyse the system.

## **Tracta**

Tracta is a CRA approach that follows the automata theoretic approach to program verification. In order to integrate analysis with other activities of software development, Tracta uses software architecture to direct analysis. In general, the software architecture of a distributed system has a hierarchical structure (Magee, et al. 94). Therefore in Tracta, the compositional hierarchy is not introduced solely for the purpose of analysis. Rather, it is directly extracted from the architectural description of the system under development.

To deal with intermediate state explosion, Tracta supports both user-specified and automatically generated contextual interfaces. This part of the Tracta approach is not presented in this paper, as it is extensively discussed in (Cheung and Kramer 96b). In Tracta, properties are introduced into analysis as finite state automata. Properties are separated into two classes, safety and liveness, with different checking mechanisms for each. The property automata specified by the user are transformed and included in the compositional hierarchy of the system. Properties do not interfere with the system behaviour, unless they are violated, in which case they are detected. Tracta has the power to validate multiple properties at the same time. The checking mechanisms of Tracta have a number of advantages. Firstly, action hiding is independent of property checking. This means that properties may involve actions that are internal to subsystems, even though these actions are not visible at the global system level. Moreover, in the presence of violations, the properties introduced may further reduce the size of both the subsystems, and the global system analysed.

## **Related work**

A number of tools have been developed that support the analysis approaches discussed in this section. SPIN (Holzmann 91) performs reachability analysis on a set of processes specified as labelled transition systems in the specification language PROMELA. SPIN checks that a program satisfies properties expressed as Büchi automata. Properties can also be expressed in linear time temporal logic, in which case they are automatically translated into automata. The SPIN system uses partial-order reduction techniques to control state explosion. The Symbolic Model Verifier SMV (McMillan 93) checks that a system satisfies properties expressed in the branching time temporal logic CTL. The tool uses binary decision diagrams (BDDs) to represent state transition systems efficiently. This representation increases the size of the systems that can be verified. The SPIN and SMV systems do not perform analysis in a compositional way.

The CADP tool-set (Fernandez, et al. 96) is a collection of tools for analysing LOTOS programs. LOTOS programs are translated into state transition systems for verification. CADP contains tools that support symbolic

representation of state machines, computation of bisimulations for comparing finite state machines, verification of properties in various temporal logics, as well as compositional state space generation. To control intermediate state explosion, a semi-composition operator is used that restricts the behaviour of intermediate subsystems according to contextual interfaces (Krimm and Mounier 97).

The FDR tool (Roscoe 94) checks that a CSP program (Hoare 85) refines its specifications specified as CSP processes. The Concurrency Workbench (Cleaveland, et al. 93b) is an automated tool for analysing networks of finite processes expressed in CCS (Milner 89). Its verification capabilities include equivalence checking between CCS processes, and the checking of properties expressed in a modal logic based on the propositional  $\mu$ -calculus. FDR and the Concurrency Workbench support minimisation of state machines, which may be used for performing compositional state space generation. However, this possibility is not made explicit, and the problem of intermediate state explosion is not addressed.

In all of the tools mentioned above that support CRA, properties can be checked only if they involve the globally observable behaviour of the system. As discussed, Tracta provides techniques that have the unique feature of allowing properties to contain internal actions of subsystems.

### Case study: the RMTP protocol

Throughout the paper we illustrate our approach using a Reliable Multicast Transport Protocol (RMTP) (Lin and Paul 96). The protocol is designed for applications that cannot tolerate data loss. It provides sequenced, loss-less delivery of data from a sender to a group of receivers, at the expense of delay. Reliability is achieved by a periodic transmission of acknowledgements by the receivers and a selective retransmission mechanism by the sender. For scalability, receivers are grouped into a hierarchy of local regions, with a *Designated Receiver* (DR) in each of those regions. Receivers in each local region send their acknowledgements to the corresponding DR, DRs send their acknowledgements to the higher level DRs or to the sender (Figure 1), thereby avoiding the acknowledgement implosion problem. DRs cache received data and are in charge of retransmissions within their local regions, thus decreasing end-to-end latency. The term *Acknowledgement Processor* (AP) is used to denote either a designated receiver or the sender, when referring to them as entities that receive and process acknowledgements. Receivers that are not designated receivers are referred to as *ordinary receivers*.

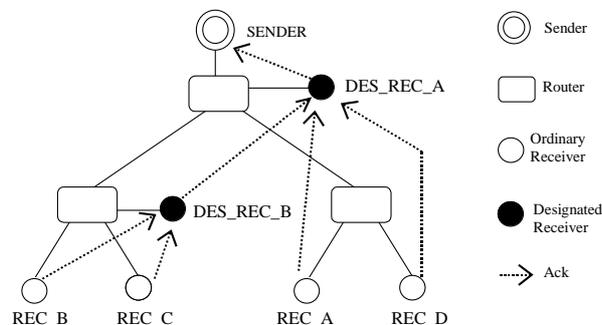


Figure 1. A multicast tree of receivers

To cater for situations where designated receivers may fail, receivers use a mechanism to dynamically select the nearest operational AP in the multicast tree. This is the part of the RMTP protocol on which our case study focuses. In RMTP, dynamic selection of APs is achieved by the use of a special packet, called the SEND\_ACK\_TO\_ME packet. The sender and all DRs periodically advertise themselves by multicasting SEND\_ACK\_TO\_ME packets along their sub-trees. These packets are tagged with the same initial TIME\_TO\_LIVE values. Routers decrement the TIME\_TO\_LIVE value when forwarding packets. Therefore a larger TIME\_TO\_LIVE value indicates a closer proximity in the multicast tree. On receipt of a SEND\_ACK\_TO\_ME packet, a receiver compares the TIME\_TO\_LIVE value associated with the incoming packet with that associated with the AP currently selected. The receiver switches to a new AP if the incoming packet has a larger TIME\_TO\_LIVE value. When a receiver fails to receive a new SEND\_ACK\_TO\_ME packet from the currently selected AP after a certain period of time, it assumes failure of the AP and initiates a new selection round.

## Paper structure

This paper presents an overview of the Tracta approach. It places particular emphasis on describing the way in which the software architecture of a distributed system can be fruitfully exploited for the analysis of the system behaviour. The remainder of the paper is organised as follows. Section 2 presents the way in which system behaviour is formally specified in Tracta, based on the software architecture. Section 3 focuses on the expression and checking of safety and liveness properties. Section 4 briefly describes the Tracta tool in terms of its environment, its main modules and the underlying algorithms. In all sections, we illustrate the concepts presented by means of the RMTP case study. Finally, in section 5, we conclude with a discussion and plans for future work.

## 2. Tracta as a method

### 2.1 Description of software architectures

Software architecture has been identified as a promising approach to bridging the gap between requirements and implementations in the design of complex systems. Software architecture describes the gross organisation of a system in terms of its components and their interactions. Darwin (Magee, et al. 95) is an architecture description language that has been extensively used for specifying the structure of distributed systems and subsequently directing their construction (Magee, et al. 95, Magee, et al. 94). It has both a textual and a graphical syntax with appropriate tool support (see section 4.1). Darwin describes a system in terms of components that manage the implementation of services. In general, systems have a hierarchical structure, as their components can themselves be defined as instances of composite component types with substructure.

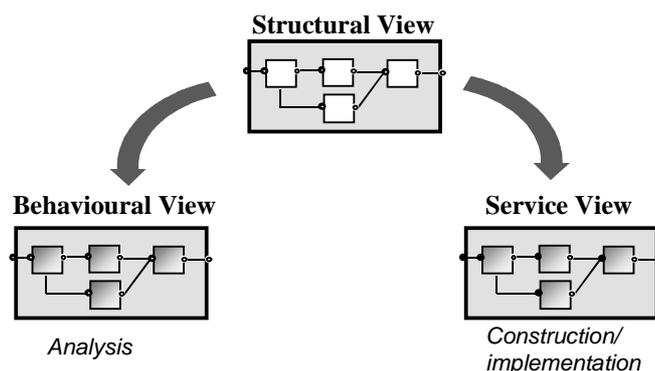


Figure 2. Common structural view with service and behavioural views

Darwin is a language that supports multiple views, two of which are the behavioural view (for behaviour analysis) and the service view (for construction) (Figure 2). Each view is an elaboration of the basic structural view: the skeleton upon which we hang the flesh of behaviour specification or service implementation (Kramer and Magee 97, Magee, et al. 97). In this section we use the RMTP case study to illustrate how a behavioural model can be produced based on the Darwin description of software architecture.

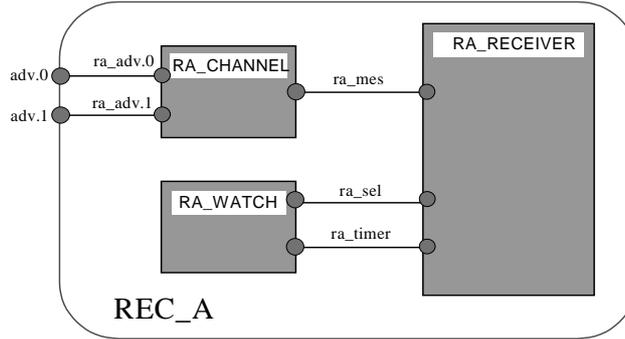
In the RMTP case study, each ordinary and designated receiver is further decomposed into three primitive components (Figure 4). The first component implements the main functionality of the receiver, which is the dynamic selection of an acknowledgement processor (e.g. component RA\_RECEIVER for REC\_A and component DRA\_DES\_RECVR for DES\_REC\_A). The second component implements the timeouts that initiate the selection of a new acknowledgement processor (e.g. components RA\_WATCH and DRA\_DES\_WTCH). The third component implements an unreliable channel that may lose messages (e.g. components RA\_CHANNEL and DRA\_DES\_CHNL). Note that the above components can be obtained by instantiating the appropriate component types. However, since analysis is performed on an instantiated software architecture, we only discuss component instances in the rest of this paper.

Consider ordinary receiver REC\_A in Figure 1, for example. Figure 3 presents the architectural description of component REC\_A, which encapsulates the three components. In Darwin, communicating actions take place

where portals of components (represented as grey dots) are bound together. A *portal* is an interface instance and has a type that (at this structure level) is simply a set of names. These names refer to actions or events shared between bound components. For example, interface `ra_timer` in Figure 3 is a composite interface instance of type `TIMER_OPER`, defined in Darwin as follows:

```
portal ra_timer:TIMER_OPER
interface TIMER_OPER {tmout; reset;}
```

Portals can be given local names, however binding is allowed only between portals of the same type. In Darwin, a component interacts with its environment through an *external interface*, and encapsulates all interactions among its sub-components that do not form part of this interface. The external interface of component `REC_A` consists of the portals `adv.0` and `adv.1`.



**Figure 3. Structural view of the ordinary receiver `REC_A` in Darwin**

The architectural hierarchy of components of the RMTP illustrated in Figure 4 reflects the multicast tree of Figure 1. Receivers `REC_C` and `REC_D` have not been included in the case study, as they exhibit identical behaviour to that of `REC_B` and `REC_A`, respectively. All grey-coloured components in the figure represent properties for analysis, as discussed in Section 3. Finally components `RB_STBLY` under `REC_B`, and `INTERM`, have been added to the basic architecture for analysis purposes.

## 2.2 Attaching behaviour to software architectures

In general, the software architecture of a system in Darwin has a hierarchical structure, with primitive components at the leaves and composite components at the nodes of the hierarchy. In Tracta behaviour is formally modelled in terms of labelled transition systems. More specifically, behaviour is attached to the software architecture by specifying a labelled transition system for each primitive component in the hierarchy. The behaviour of composite components is computed from that of their constituent parts. For this process, all necessary information related to the structure and interconnections of components is extracted from the architectural description of the system.

*Labelled transition systems* (LTS) can be used to model the behaviour of communicating processes in a distributed program. An LTS contains all the reachable states and executable transitions of a process. The model has been widely used in the literature for specifying and analysing distributed programs (Ghezzi, et al. 91, Kempainen, et al. 92, Rabinovich 92, Valmari 92).

Let  $States$  be the universal set of states,  $L$  be the universal set of observable action labels, and  $Act = L \cup \{\tau\}$ , where  $\tau$  is used to denote an action that is internal to a subsystem, and therefore unobservable by its environment. An LTS of a process  $P$  is a quadruple  $\langle S, A, \Delta, q \rangle$  where

- (i)  $S \subseteq States$  is a set of states;
- (ii)  $A = \alpha P \cup \{\tau\}$ , where  $\alpha P \subseteq L$  denotes the communicating *alphabet* of  $P$ ;
- (iii)  $\Delta \subseteq S \times A \times S$ , denotes a transition relation that maps from a state and an action onto another state;

(iv)  $q \in S$  indicates the initial state of  $P$ .

Since there is a one-to-one mapping between a process  $P$  and its LTS, we use the term process and LTS interchangeably.

We say that an LTS  $\langle S, A, \Delta, q \rangle$  transits with action  $a \in A$  into another LTS  $\langle S, A, \Delta, q' \rangle$  if  $(q, a, q') \in \Delta$ . That is,

$$\langle S, A, \Delta, q \rangle \xrightarrow{a} \langle S, A, \Delta, q' \rangle \text{ if } (q, a, q') \in \Delta.$$

A process  $P = \langle S, A, \Delta, q \rangle$  is *deterministic* if  $\forall s, s', s'' \in S, ((s, a, s') \in \Delta \wedge (s, a, s'') \in \Delta) \Rightarrow s' = s''$ , otherwise it is *non-deterministic*. A *trace* of a process  $P$  is a sequence of observable actions that  $P$  can perform starting from its initial state (Hoare 85). We denote the set of possible traces of a process  $P$  as  $tr(P)$ .

The behaviour of a composite component is obtained from the concurrent execution of its constituent parts. In terms of LTS, a concurrent system is described by the parallel composition of its component processes. Formally, the *parallel composition* operator  $\parallel$  is a binary operator. It takes two LTSs  $P = \langle S_1, A_1, \Delta_1, q_1 \rangle$  and  $Q = \langle S_2, A_2, \Delta_2, q_2 \rangle$  as arguments, and returns the LTS  $P \parallel Q = \langle S_1 \times S_2, A_1 \cup A_2, \Delta, (q_1, q_2) \rangle$ , where **Rule 1** describes the transitional semantics of  $\Delta$ :

**Rule 1:** Let  $a \in Act$ . Then:

$$\frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q} \quad a \notin \alpha Q \qquad \frac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'} \quad a \notin \alpha P$$

$$\frac{P \xrightarrow{a} P'; Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P' \parallel Q'} \quad a \neq \tau$$

The parallel composition operator is both commutative and associative. Therefore the order in which processes are composed is insignificant. According to the composition operator described, processes communicate by synchronisation on actions common to their alphabets with interleaving of the remaining actions. Modelling of primitive components is therefore sensitive to the selection of action names. Global name sensitivity is impractical in distributed systems where component specifications may be reused or may have been developed independently. We therefore need to equip the LTS model with operators that reflect the notion of *external interfaces* as well as *bindings* between Darwin interfaces.

In Tracta, a component defines a scope for the actions in its behaviour. To make this component communicate with its environment, a *relabelling* operator is necessary. Binding portals in Darwin corresponds to *relabelling* with common names the corresponding actions in LTSs. According to the semantics of the composition operator, these actions are then forced to execute synchronously. In Darwin, a component interacts with its environment through an *external interface*, and encapsulates all interactions among its sub-components that do not form part of this interface. In terms of LTSs, actions in component behaviour are *hidden* (i.e. made unobservable) unless they belong to the external interface of the component. The hiding and relabelling operators supported by Tracta are formally defined as follows.

The *hiding* operator  $\hat{\uparrow}$  takes as arguments an LTS  $P = \langle S_1, A_1, \Delta_1, q_1 \rangle$  and a set  $M \in L$  and returns the LTS  $P \hat{\uparrow} M = \langle S_1, (A_1 \cap M) \cup \{\tau\}, \Delta, q_1 \rangle$ , where the semantics for  $\Delta$  is given by **Rule 2** below:

**Rule 2:** Let  $a \in Act$ . Then:

$$\frac{P \xrightarrow{a} P'}{P \hat{\uparrow} M \xrightarrow{a} P' \hat{\uparrow} M} \quad a \in M \qquad \frac{P \xrightarrow{a} P'}{P \hat{\uparrow} M \xrightarrow{\tau} P' \hat{\uparrow} M} \quad a \notin M$$

The *relabelling* operator  $/$  takes an LTS  $P = \langle S_I, A_I, \Delta_I, q_I \rangle$  and a function  $f: Act \longrightarrow Act$  such that  $f(\tau) = \tau$ , and returns the LTS  $P/f = \langle S_I, f(A_I), \Delta, q_I \rangle$ , where the semantics for the transition relation of  $P/f$  is given by **Rule 3** below:

**Rule 3:** Let  $a \in Act$ . Then:

$$\frac{P \xrightarrow{a} P'}{P/f \xrightarrow{f(a)} P'/f}$$

Finally, we wish to be able to express the fact that any component  $C$  in a system can be substituted by another component  $C'$ , on condition that  $C$  and  $C'$  have the same external interface and that their behaviours in terms of this interface are indistinguishable. This is reflected in our model by equating LTSs that are equivalent with respect to *weak semantic (or observational) equivalence* (Milner 89). Weak semantic equivalence has the desirable property of being a congruence (Milner 89) with respect to the operators used in LTS expressions. It equates systems that exhibit the same behaviour to the external observer who cannot realise the occurrence of  $\tau$ -actions.

Formally, let  $\wp$  be the universal set of LTS, and  $P \stackrel{\alpha}{\approx} P'$  denote  $P \xrightarrow{\tau^* a \tau^*} P'$ , where  $\tau^*$  means zero or more  $\tau$ s. Then *weak semantic equivalence*  $\approx$  is the union of all relations  $R \subseteq \wp \times \wp$  satisfying that  $(P, Q) \in R$  implies:

1.  $\alpha P = \alpha Q$ ;
2.  $\forall a \in L \cup \{\varepsilon\}$ , where  $L = Act - \{\tau\}$ , and  $\varepsilon$  is the empty sequence:
  - $P \stackrel{\alpha}{\approx} P'$  implies  $\exists Q', Q \stackrel{\alpha}{\approx} Q'$  and  $(P', Q') \in R$ .
  - $Q \stackrel{\alpha}{\approx} Q'$  implies  $\exists P', P \stackrel{\alpha}{\approx} P'$  and  $(P', Q') \in R$ .

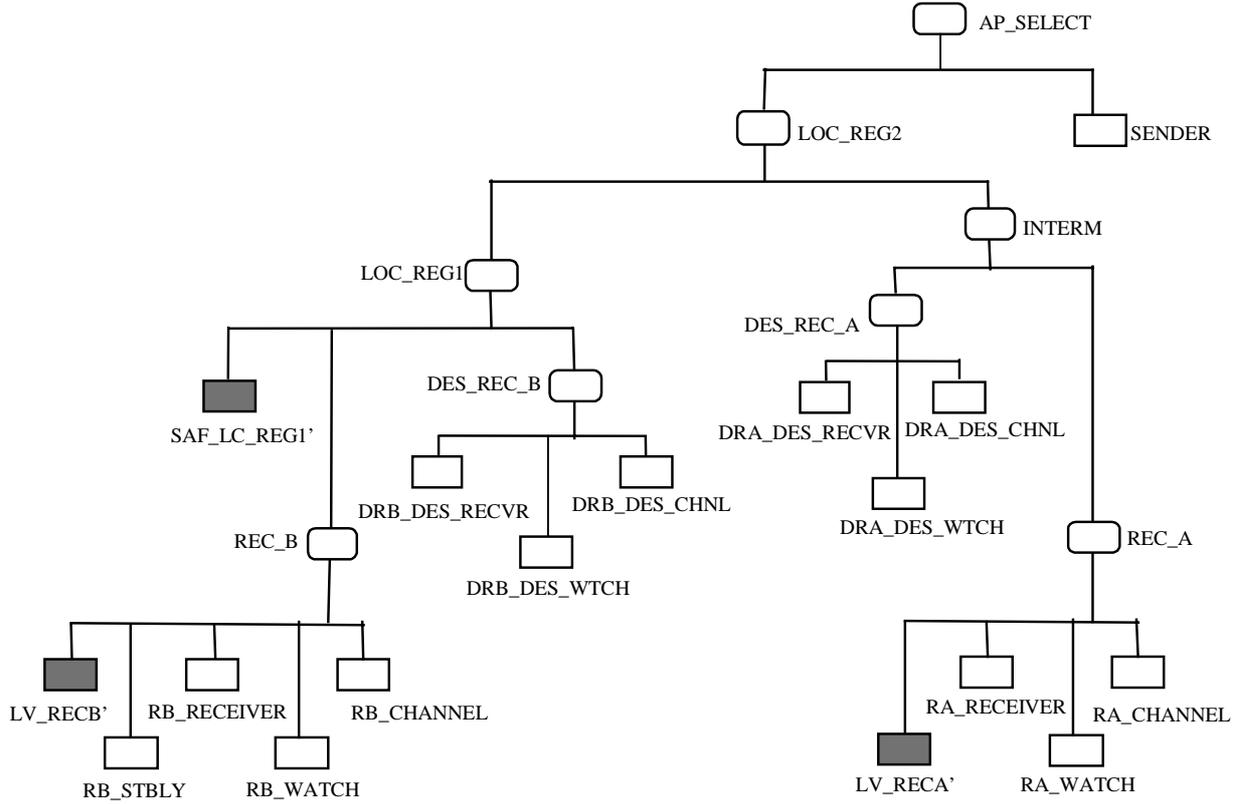
## Summary

In Tracta, the hierarchy of the software architecture is used both for deriving and analysing the behaviour of the system in stages. Each primitive component is associated with behaviour explicitly expressed as an LTS. The behaviour of every composite component is described by the parallel composition of the LTSs of its sub-components, where relabelling is applied based on the bindings between these sub-components in the software architecture. The LTS of the composite component can therefore be computed and subsequently checked against local properties. The external interface of the component also determines which actions in the LTS must be hidden. The resulting LTS is minimised with respect to weak semantic equivalence, before being used for the computation of more composite components.

## 2.3 Modelling the behaviour of components of RMTP

In our case study, we have modelled and analysed the part of the RMTP protocol that deals with dynamic selection of acknowledgement processors for the multicast tree depicted in Figure 1. In this section we make a detailed presentation of the way we have modelled the behaviour of an ordinary receiver, REC\_A, and a designated receiver, DES\_REC\_A. We have chosen to discuss these, as their behaviour is both representative of that of other components in the system, but also simple enough to allow a clear graphical illustration (see Figures 5 and 6).

In the RMTP, all receivers initially select the sender as their acknowledgement processor. This is reflected in the initial states of the LTSs that describe their behaviour. Three processes are associated with each ordinary and designated receiver in the multicast tree. As illustrated in Figure 4, the behaviour of ordinary receiver REC\_A is described by the parallel composition of components RA\_RECEIVER, RA\_WATCH and RA\_CHANNEL. Actions in all three components of REC\_A are prefixed with “ra\_” to differentiate them from actions of other components. This is to reflect the fact that a component defines a scope for actions in its behaviour. Relabelling can then be applied so as to make specific actions synchronise.



**Figure 4. Compositional hierarchy for the RMTP**

The RA\_CHANNEL process models a lossy channel that receives advertisements from acknowledgement processors above the receiver (actions  $ra\_adv.0$ ,  $ra\_adv.1$ , where in general  $0$  represents the sender and  $1$  represents DES\_REC\_A). It non-deterministically decides to transmit them to RA\_RECEIVER (actions  $ra\_mes.0$ ,  $ra\_mes.1$ ), or lose them. The specification assumes fair execution in the sense that unfair execution sequences, where RA\_CHANNEL keeps losing all messages, are ignored.

In general, we model channels as non-deterministic processes. This is because we want to express the fact that a channel may be in a state where it has committed to transmit. This will permit detection of possible deadlocks, which might be concealed with a deterministic model where losing a message is always a legal alternative. Moreover channels have a capacity of one message. We have avoided adding buffers in our example as they can significantly complicate and increase the size of the case study. Buffers are not necessary because the channels can always deal with the messages they contain and then be ready to receive new messages. As a result, the capacity of one does not introduce any deadlocks.

Process RA\_WATCH models the time-out associated with the selection of a new acknowledgement processor (AP). Whenever an AP is selected by RA\_RECEIVER ( $ra\_sel.0$ ,  $ra\_sel.1$ ), RA\_WATCH sets a timer ( $ra\_set\_tmout$ ) after which stage a timeout may occur. As illustrated in Figure 3, action  $ra\_set\_tmout$  does not belong to the external interface of component RA\_WATCH and is therefore substituted by the internal  $\tau$  action. When RA\_RECEIVER receives a message from its currently selected AP, it resets the timer ( $ra\_timer.reset$ ). When a timeout occurs ( $ra\_timer.tmout$ ), RA\_RECEIVER assumes that its current AP has failed. It moves to state NO\_AP (no acknowledgement processor currently selected) and the selection of a new AP is initiated.

Process RA\_RECEIVER is in charge of selecting an AP. When it is in state NO\_AP, it selects the AP whose advertisement it receives first ( $ra\_mes.0$ ,  $ra\_mes.1$ ). The currently selected AP is changed whenever an advertisement is received from a nearer AP (see transition (DR0,  $ra\_mes.1$ , SEL1) in Figure 5). Based on the Darwin structure of REC\_A (Figure 3), its behaviour is described by the following expression:

$$REC\_A = (RA\_CHANNEL \parallel RA\_WATCH \parallel RA\_RECEIVER) / \{adv.0/ra\_adv.0, adv.1/ra\_adv.1\} \uparrow \{adv.0, adv.1\}.$$

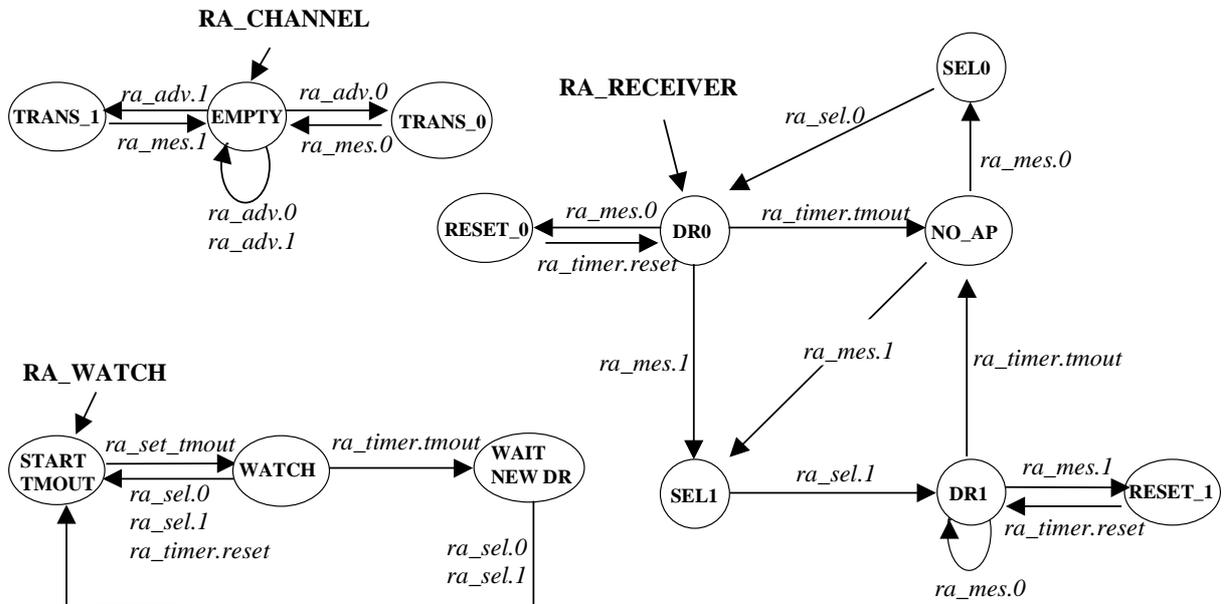


Figure 5. LTS of the receiver REC\_A

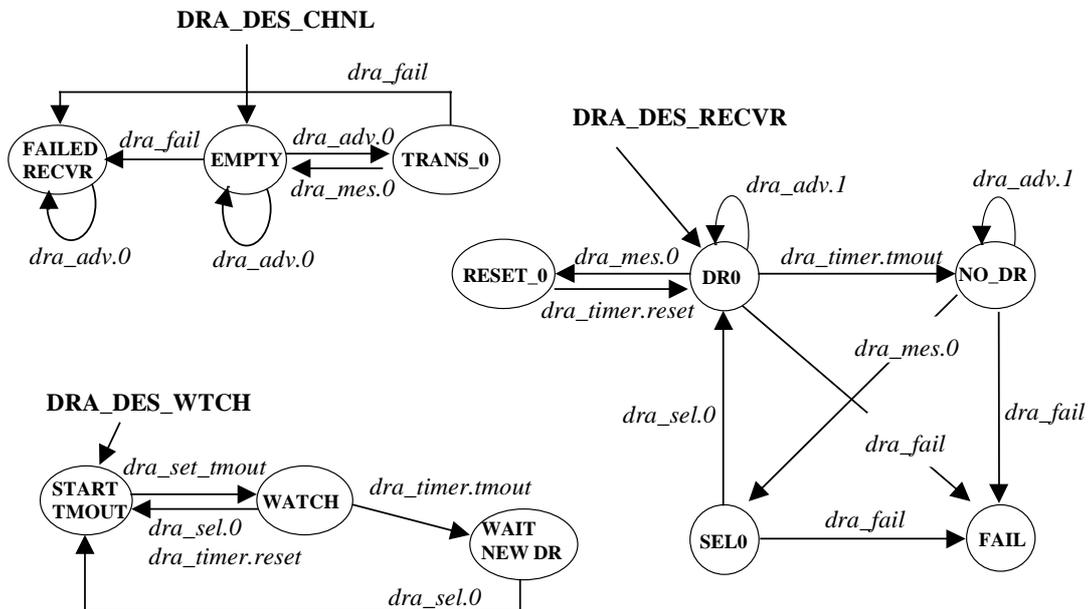


Figure 6. LTS of the designated receiver DES\_REC\_A

All advertisements  $ra\_adv^*$  are relabelled to  $adv^*$ , to make them synchronise with the corresponding actions of other components. These actions constitute the external interface of REC\_A. The behaviour of REC\_B is modelled in a similar way, with the difference that it can additionally receive messages from DES\_REC\_B. For analysis purposes, we have added component RB\_STBLY to REC\_B (see Figure 4). This component simply records the fact that RB\_RECEIVER is stable, i.e. that it is in a state where it has a currently selected acknowledgement processor. All stable states  $s$  in REC\_B are thus distinguished by the existence of a  $(s, rb\_stable, s)$  transition. We do not hide action  $rb\_stable$  in our case study.

Figure 6 illustrates the behaviour of designated receiver DES\_REC\_A. Designated receivers behave like ordinary receivers, except that they may fail and advertise themselves (for DES\_REC\_A, actions  $dra\_fail$ ,  $dra\_adv.1$ , respectively). In our case study, we do not hide actions related to the failure of designated receivers. Although these actions are not part of the external interface, we want them to be visible at the global system level. The behaviour of DES\_REC\_A is expressed by the following expression:

$$DES\_REC\_A = (DRA\_DES\_CHNL \parallel DRA\_DES\_WTCH \parallel DRA\_DES\_RECVR) / \{adv.0/dra\_adv.0, adv.1/dra\_adv.1\} \uparrow \{adv.0, adv.1, dra\_fail\}.$$

Note that we have not modelled failure for ordinary receivers and the sender. If the sender fails, the multicast session is cancelled, in which case RMTP does not need to fulfil its objectives. Properties of the ordinary receivers are not expected to hold when the latter fail. Moreover, failures of ordinary receivers do not affect the behaviour of their environment, and may therefore be ignored. Routers have not been specified as separate processes because our model directly supports multicast by the synchronisation of actions that are common to the process alphabets.

### 3. Expressing and checking properties

In model checking, a model for an abstraction of a system is verified against a set of properties that constitute the system specifications. Tracta checks that a system, described as a collection of communicating LTSs organised in a hierarchical structure, satisfies its required properties. Properties are separated into two classes: *safety* and *liveness* (Manna and Pnueli 92, Andrews 91). A *safety* property asserts that the program never enters an undesirable state. For example, mutual exclusion is a safety property that specifies the absence of a program state where a common resource is simultaneously accessed by more than one client. A *liveness* property asserts that a program eventually enters a desirable state. For example, the assertion that a program will eventually close a file after opening it, is a liveness property.

In the LTS model for system behaviour, characteristics of (groups of) states can only be distinguished in terms of action scenarios (sequences of transitions), the occurrence of which guarantee these characteristics for the states reached. In such settings, the classification of properties into safety and liveness can be realised as follows. Safety properties express the fact that subsequent to specific scenarios, the occurrence of some actions must be prevented if undesirable system states are to be avoided. On the other hand, liveness properties ensure progress in a system by enforcing the eventual occurrence of actions following specific scenarios. In this context, scenarios express the conditions that make these eventualities necessary.

In this section, we present how properties are expressed in Tracta, as well as the checking mechanisms that have been introduced in the method in order to specifically address issues related to CRA techniques.

#### 3.1 Limitations of the conventional approach to compositional reachability analysis

Promising results have been reported in the literature on the use of a compositional approach to derive the overall system behaviour using reachability analysis (Sabnani, et al. 89, Tai and Koppol 93, Yeh 93a). In compositional reachability analysis (CRA) techniques, the model of the target system is given as an LTS that describes an abstraction of the system behaviour, according to the requirements of the user. As described, the analysis proceeds in a bottom-up manner by gradual composition of the LTS of the overall system from those of its subsystems. At each intermediate step, the LTS of a subsystem is simplified by hiding internal actions and subsequently minimising the LTS with respect to observational equivalence.

The key to the success of CRA techniques is to employ a modular software architecture and hide as many internal actions as possible in each subsystem, thereby producing a simpler LTS in general. However, the properties that are available for reasoning in the analysis are then constrained by the set of remaining globally observable actions. In order to check properties that may involve actions internal to subsystems, these actions must be exposed at the global graph of the system. This compromises the CRA approach. The objective is to retain the freedom of abstracting (by hiding) (sub)system behaviour at the various levels of the system hierarchical structure, without compromising the effectiveness of analysis.

One of the main contributions of the Tracta approach to compositional methods is the following: for both safety and liveness properties, Tracta provides checking mechanisms that are independent from the actions that remain globally observable in the system state graph (Cheung, et al. 97, Cheung and Kramer 96a). These mechanisms rely on well-defined transformations that are applied to the properties specified by the system developer. All transformations are performed automatically and are therefore transparent to the user.

### 3.2 Safety properties

In Tracta, a safety property is specified by the user as a deterministic LTS  $P = \langle S, A, \Delta, p \rangle$  that contains no  $\tau$ -transitions. Property  $P$  exclusively reasons about occurrences of actions that belong to its alphabet  $A$ . More specifically, the property defines the set  $T$  of acceptable behaviours over  $A$  as follows:  $T = \{w \in A^* \mid w \in tr(P)\}$ . Let  $Sys$  be a system to be verified against property  $P$ .  $Sys$  will then be said to satisfy  $P$  if and only if  $tr(Sys \uparrow \alpha P) \subseteq tr(P)$ . Informally, a system  $Sys$  satisfies a property  $P$  if  $Sys$  can only generate traces which, when restricted to the alphabet of  $P$ , are acceptable to  $P$ .

In order to capture undesirable traces of system behaviour, we have introduced into our model a trap state  $\pi$  with the following semantics: a process that enters state  $\pi$  transits into process  $\Pi = \langle \{\pi\}, Act, \{\}, \pi \rangle$ , which can potentially perform any action but never actually does (similar to process  $STOP_{Act}$  in CSP (Hoare 85)). Process  $\Pi$  exhibits non-standard behaviour in the context of composition and hiding, expressed by the following definitions (these definitions overload **Rules 1-2** that describe the transitional semantics of operators  $\parallel$  and  $\uparrow$ , respectively).

- $\forall Q \in \wp$  ( $\wp$  is the universal set of LTSs),  $Q \parallel \Pi =_{\text{def}} \Pi \parallel Q =_{\text{def}} \Pi$
- $\forall M \subseteq L$ ,  $\Pi \uparrow M =_{\text{def}} \Pi$

Finally, process  $\Pi$  must be distinguished from any other process in  $\wp$ . This is performed by adding to the definition of weak semantic equivalence (section 2.2), condition “3.  $(P, Q) \in \mathbf{R}$  implies  $(P = \Pi \text{ iff } Q = \Pi)$ ”.

Tracta uses the following mechanism for checking that a property  $P = \langle S, A, \Delta, p \rangle$  is satisfied by a system  $Sys$ .  $P$  is automatically converted into its *image process*  $P' = \langle S \cup \{\pi\}, A, \Delta', p \rangle$ , where  $\Delta'$  is defined as follows:

- $\Delta' = \Delta \cup \{(s, a, \pi) \mid s \in S, a \in A, \text{ and } \nexists s' \in S: (s, a, s') \in \Delta\}$ .

$P'$  is then inserted in the compositional hierarchy of the system  $Sys$ , to be composed into the (sub)system to which property  $P$  refers. Informally, the image process  $P'$  is obtained by replacing any undefined transition in  $P$  with a transition to  $\pi$ . This means that  $P'$  monitors the behaviour of the (sub)system with which it is composed, without interfering with it. However, if the behaviour is not acceptable to property  $P$ ,  $P'$  forces the subsystem into the trap state, which records the fact that a violation has occurred and prunes all behaviours subsequent to the violation.

We have proven (Cheung and Kramer 96a, Giannakopoulou 95) that properties thus introduced into analysis are violated if and only if  $\pi$  is a reachable state in the LTS of the global system. This result is not affected by the hiding of actions at intermediate stages of CRA. Tracta therefore reduces the checking of safety properties to a  $\pi$  reachability problem. Moreover, in the absence of violations, the global LTS obtained for the system under analysis is observationally equivalent to that which would have been obtained if image properties had not been introduced into the compositional hierarchy.

An important benefit of the Tracta approach to safety property checking is that it may significantly reduce the size of both intermediate and the final LTSs for a system, in the presence of property violations. This is because Tracta does not explore the reachable states of the system that follow the violation of a safety property. A limitation of the approach is that the violation of any property is mapped to the same state  $\pi$ . This means that when multiple properties have been introduced into analysis, it may be difficult to identify which properties have been violated in the global LTS of the system.

When the  $\pi$  state is reachable, a useful kind of diagnostic information consists of providing a trace of the system leading to the violating state. This trace often identifies the property that has been violated. For a more detailed presentation of the approach to safety property checking the interested reader is referred to (Cheung and Kramer 96a).

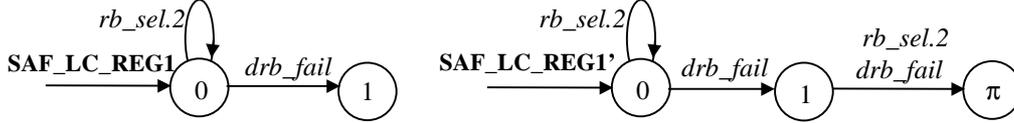


Figure 7. Safety property in RMTP

In the RMTP case study, we have included safety property SAF\_LC\_REG1 that refers to component LOC\_REG1. The property states that subsequently to the failure of DES\_REC\_B, no receiver in its sub-tree (REC\_B in our case study) selects DES\_REC\_B as its acknowledgement processor. Figure 7 illustrates property SAF\_LC\_REG1 together with its corresponding image process, which has been included in the compositional hierarchy as depicted in Figure 4. Note that action  $rb\_sel.2$  does not form part of the communication interface of component REC\_B. However, since it is used in the expression of property SAF\_LC\_REG1 of subsystem LOC\_REG1, its hiding must be postponed until the next level of the hierarchy of Figure 4.

### 3.3 Liveness properties

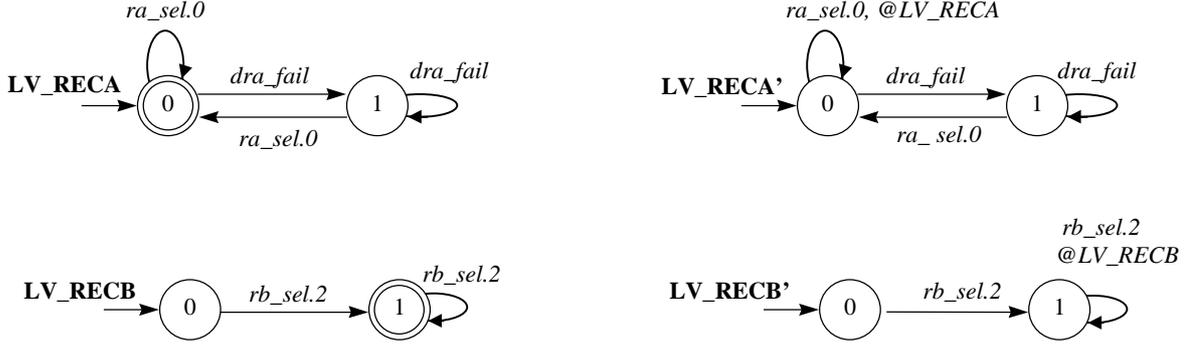
Liveness properties reason about infinite traces of a system. They are therefore specified in terms of *Büchi automata* - finite state machines that accept infinite words. It has been proven that the expressive power of Büchi automata is larger than that of linear time temporal logic (LTL) (Gribomont and Wolper 89). In general, any formula of LTL can be automatically translated into a Büchi automaton (Gerth, et al. 95, Gribomont and Wolper 89).

A Büchi automaton  $B$  is a 5-tuple  $\langle S, A, \Delta, q_0, F \rangle$ , where  $S$  is a finite set of states,  $A$  is a set of actions,  $\Delta \subseteq S \times A \times S$  is a set of transitions,  $q_0 \in S$  is the initial state, and  $F \subseteq S$  is a set of accepting states. ■

An *execution* of  $B$  on an infinite word  $w = a_0a_1a_2\dots$  over  $A$  is an infinite sequence  $\sigma = q_0q_1q_2\dots$  of elements of  $S$ , where  $(q_i, a_i, q_{i+1}) \in \Delta, \forall i \geq 0$ . For infinite words that include actions outside  $A$ , we extend the definition of an execution as follows. An execution of  $B$  on an infinite word  $w = a_1a_2a_3\dots$  over  $A_1 \supseteq A$  is an infinite sequence  $\sigma = q_0q_1q_2\dots$  of elements of  $S$ , where:

$$\forall i \geq 0, ((q_i, a_i, q_{i+1}) \in \Delta \text{ if } a_i \in A) \text{ and } (q_i = q_{i+1} \text{ if } a_i \notin A).$$

In other words, at any state  $s$  of  $B$ , the effect of an action  $a \notin A$  on some execution of  $B$  is the same as if transition  $(s, a, s)$  belonged to  $\Delta$ . An execution of  $B$  is *accepting* if it contains some accepting state of  $B$  an infinite number of times.  $B$  accepts a word  $w$  if there exists an accepting execution of  $B$  on  $w$ .



**Figure 8. Liveness properties in RMTP**

In our RMTP case study, a desirable feature of the dynamic selection mechanism is that it is always the case that, upon failure of a designated receiver, all receivers in its sub-tree eventually select a different acknowledgement processor. For component REC\_A this property reduces to checking that if DES\_REC\_A fails (*dra\_fail*), then REC\_A is eventually able to select SENDER as its acknowledgement processor (*ra\_sel.0*). The property is expressed as  $\square(dra\_fail \Rightarrow \Diamond ra\_sel.0)$  in the linear temporal logic LTL (Gribomont and Wolper 89). Büchi automaton LV\_RECA in Figure 8 expresses this property for REC\_A. Since state 0 is the accepting state of LV\_RECA, an infinite word is accepted by LV\_RECA if there exists an execution of the automaton in which state 0 appears infinitely often. Therefore LV\_RECA accepts the language  $(dra\_fail^* ra\_sel.0)^\omega$ , where juxtaposition represents concatenation, and the operators  $*$  and  $^\omega$  denote finite and infinite repetition, respectively.

Note that property LV\_RECA requires the eventual occurrence of action *ra\_sel.0*, even in the situation where SENDER is the currently selected AP of REC\_A when action *dra\_fail* occurs. As discussed later in this section, the liveness checking mechanism used in our case study assumes fair choice on system transitions. As a result, even when SENDER is the selected AP of REC\_A, time-out expiration eventually happens and initiates a new round of AP selection. Such time-out expirations reflect a delay in the receipt of advertisement messages sent from the SENDER. If DES\_REC\_A has failed in the meanwhile, the protocol requires that *ra\_sel.0* eventually happens. Therefore LV\_RECA needs to be satisfied by our model of RMTP. A more complicated property can be used to express that “after the failure of the currently selected AP, a new AP is eventually selected” but, for simplicity, property LV\_RECA is preferred.

Another property that we have included in our case study is LV\_RECB, illustrated in Figure 8. This property refers to REC\_B and states that it is eventually the case that REC\_B selects DES\_REC\_B to be its acknowledgement processor (*rb\_sel.2*). In LTL, the property is expressed as  $(\Diamond rb\_sel.2)$ . We thereby wish to check that, although initially all receivers in the system start with the SENDER as their acknowledgement processor, each one eventually selects the one nearest to it in the multicast tree.

In the automata theoretic approach to program verification, Büchi automata are composed with the system under analysis. In the LTS system model where no specific information is stored on states, Tracta distinguishes accepting states of Büchi automata by means of special transitions that are added to the automata according to the following transformation:

**Definition 1:** A Büchi automaton  $B = \langle S, A, \Delta, q_0, F \rangle$  is mapped into a liveness property LTS  $B' = \langle S, A \cup \{ @B \}, \Delta', q_0 \rangle$  by adding a new globally unique action  $@B$  and new transitions called accepting transitions, where:

- (i)  $@B \notin \alpha A$ ; and
- (ii)  $\Delta' = \Delta \cup \{ s \xrightarrow{@B} s \mid s \in F \}$ . ■

Figure 8 illustrates the results of applying the transformation to properties LV\_RECA and LV\_RECB. The transformation is performed automatically and transparently to the user.

Tracta checks a liveness property expressed as a Büchi automaton  $B = \langle S, A, \Delta, q_0, F \rangle$ , where the following conditions hold:

1.  $B$  is deterministic (therefore a single execution of the automaton is defined on an infinite word).
2. A transition is defined at each state in  $S$  for every action in  $A$  (we use the term “ $B$  is complete” to denote this fact).
3. Fair choice is assumed in the model of the system analysed. This means that if a choice over a set of transitions is executed infinitely often, then each transition in the set is executed infinitely often.

A system satisfies a property expressed as a Büchi automaton if the automaton accepts all infinite executions of the system. In a finite state system where fair choice is assumed, infinite executions can only be obtained by the infinite repetition of states within a terminal set of states of the system, where a terminal set of states is defined as follows:

**Definition 2:** A set of states  $C$  in an LTS  $\langle S, A, \Delta, q \rangle$  is said to be *terminal* if and only if:

- a)  $C$  is a *strongly connected component*; and
- b)  $C$  is closed under  $\Delta$ , i.e.,  $\forall s \in C, (s, a, s') \in \Delta \Rightarrow s' \in C$ . ■

Under conditions 1 to 3 presented above, we can check that a system  $P$  satisfies a property expressed by  $B$  in the following way:

- Compute  $P \parallel B'$ , where  $B' = \langle S, A \cup \{ @B \}, \Delta', q_0 \rangle$  is the liveness property LTS for  $B$ .
- Check that each terminal set of states in  $P \parallel B'$  contains at least one state  $s$  such that  $(s, @B, s)$  is a transition of  $P \parallel B'$  (we then say that  $@B$  is *enabled* in the terminal set of states). If this is the case, then  $P$  satisfies the property. Computing all terminal sets of states in  $P \parallel B'$  can be performed by computing the strongly connected components of the graph, for which there exist computationally inexpensive (linear) algorithms (Tarjan 72).
- If  $P$  violates the property, return diagnostic information that will guide the developer in uncovering the error in the design. Such information consists of a trace leading to the root of the violating terminal set of states, as well as the actions that label transitions between states in the terminal set.
- If no violation is detected, remove all accepting transitions from the graph of the system and minimise. The resulting LTS is equivalent to the one that would have been obtained if liveness property LTSs had not been included into analysis.

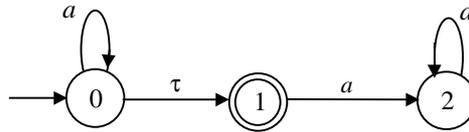
In Tracta, liveness property LTSs are composed into the (sub)system to which they refer (see Figure 4). In this way, Büchi automata may involve internal actions of the subsystem that are unobservable to the subsystem environment. According to condition 3, Büchi automata introduced into analysis are complete. This means that in exactly the same way as for safety properties, these automata simply monitor the behaviour of the system, without interfering with it. This makes it possible to check multiple liveness properties simultaneously. Since accepting transitions ( $@B$ ) identify the property to which they correspond, it is possible to detect violations of individual properties. Finally, it can be easily checked that observational equivalence preserves terminal sets of states. Therefore under fair choice, violations of liveness properties thus introduced into analysis are not lost during minimisation.

A formal proof of the mechanism presented can be found in (Giannakopoulou 98a). We summarise that the mechanism for checking liveness properties described in this section exhibits three main desirable features (Cheung, et al. 97). Firstly, it makes the hiding of actions independent of the liveness properties that are to be checked in the final graph. Secondly, it checks multiple properties simultaneously, specifically identifies the violated ones and generates the overall system behaviour. Thirdly, it avoids keeping specific information on states. Instead, states are differentiated by the actions that can be performed. As a result, no extensions are necessary to

the basic LTS model. The checking mechanism can easily be incorporated in any tool that supports compositional reachability analysis, as it requires no modifications to the composition and minimisation algorithms.

### Discussion

As mentioned, the liveness property checking mechanism presented here relies on three conditions. The fact that Büchi automata are required to be complete is not a limitation of our method. A Büchi automaton can always be made complete by adding a new state (Fernandez, et al. 92).



**Figure 9. A non-deterministic Büchi automaton**

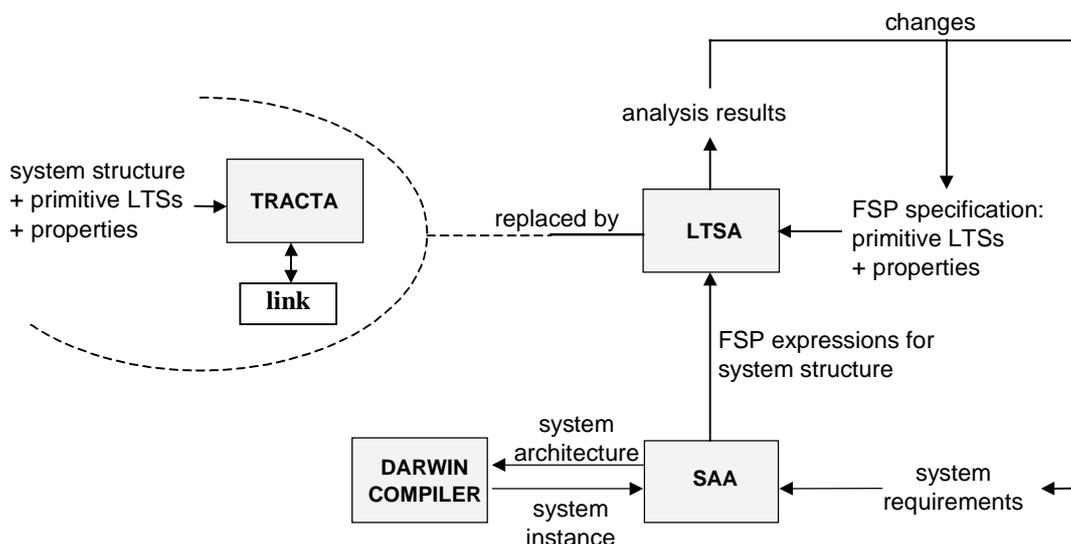
On the other hand, the fact that Büchi automata introduced in analysis need to be deterministic, imposes a restriction on the kinds of properties that may be expressed. For example, it is not possible to express with a deterministic automaton the property that, in every system execution there is a point after which action  $a$  never occurs again ( $\hat{\diamond}\Box\neg a$ ). This property is described by the automaton illustrated in Figure 9. Non-deterministic Büchi automata may have more than one execution on a word, and it is enough for one of them to be accepting for the automaton to accept the word. Therefore, when a non-deterministic liveness property LTS  $B'$  is composed with the system, the system may contain terminal sets of states where  $@B$  is not enabled, even though the system satisfies the property expressed by  $B$ .

In these cases, rather than using the automaton for a property  $F$  of interest, one needs to use the automaton  $B$  for  $\neg F$  (Aggarwal, et al. 90, Godefroid and Holzmann 93, Gribomont and Wolper 89). Then the system satisfies the property if no execution of the system is accepted by  $B$ . Again the liveness property LTS  $B'$  is composed with the system  $P$ . The system then satisfies property  $F$ , if there is no terminal set of states in  $P \parallel B'$  where action  $@B$  is enabled. Due to space limitation, we refer the interested reader to (Giannakopoulou 98a) for more details on property checking in Tracta. There, alternative mechanisms are additionally provided, which cover cases where the conditions imposed here are not required to hold. These mechanisms are put in the context of CRA where actions are hidden and subsystems minimised.

Under fair choice, the search for illegal behaviour may be too coarse grained to detect all interesting violations (Cheung, et al. 97). On the other hand, checking liveness when fairness is not taken into consideration usually returns a large number of non-realistic violations. In (Giannakopoulou 98a) fairness is discussed in detail, and various approaches are proposed for addressing the problem. The approaches range from explicit modelling of fairness in the form of constraints on the system executions (Aggarwal, et al. 90, Gribomont and Wolper 89), to efficient tests that do not involve automata and are combined with practical heuristics.

## 4. Tracta as a tool

### 4.1 The environment



**Figure 10. Tool support for design and analysis of distributed systems**

In section 2.1 we described the basic features of the Darwin architecture description language. We have also demonstrated that the description of system structure performed in this way can be utilised to provide the compositional hierarchy for analysis, and to direct hiding and relabelling as required. This integration needs to be reflected in our software tools.

The Software Architect's Assistant (SAA) (Ng, et al. 96) is a visual environment for the design and development of distributed programs using Darwin architectural descriptions. Facilities provided include the display of multiple integrated graphical and textual views, a flexible mechanism for recording design information and the automatic configuration of program code and formatted reports from design diagrams. A system architecture is used by the Darwin compiler to generate a system instance (Figure 10). The hierarchical structure of a system instance can then be utilised for analysis. The SAA is currently being re-implemented in Java to aid portability and interoperability. As shown in Figure 11, the new version of the SAA generates LTS expressions for compositional analysis, based on the system architecture ( $\hat{\uparrow}$  in the figure is an ASCII representation for the hiding operator  $\uparrow$ ).

The Tracta approach has long been supported by a tool implemented in C++. We have experimented with various ways of linking the Tracta tool to the SAA (Figure 10). These attempts were based on providing a user interface that would translate graphical input for LTS specifications and for the software architecture, into the format accepted by the Tracta tool. Our experience has shown, however, that graphical descriptions of LTSs become impractical for LTSs that involve more than a few states. We have therefore abandoned the effort of providing an interface that will link the Tracta tool to the SAA. Rather, we have moved towards building a new tool in Java, the LTSA (Kramer and Magee 97, Magee, et al. 97), to aid portability and interoperability of the whole environment (Figure 10). The LTSA already implements a large part of the Tracta approach. The new tool accepts input in a process algebra notation, called FSP for Finite State Processes. It can directly use the expressions generated by the SAA for performing compositional analysis directed by the software architecture. We are currently working on a tighter integration between the LTSA and the SAA.

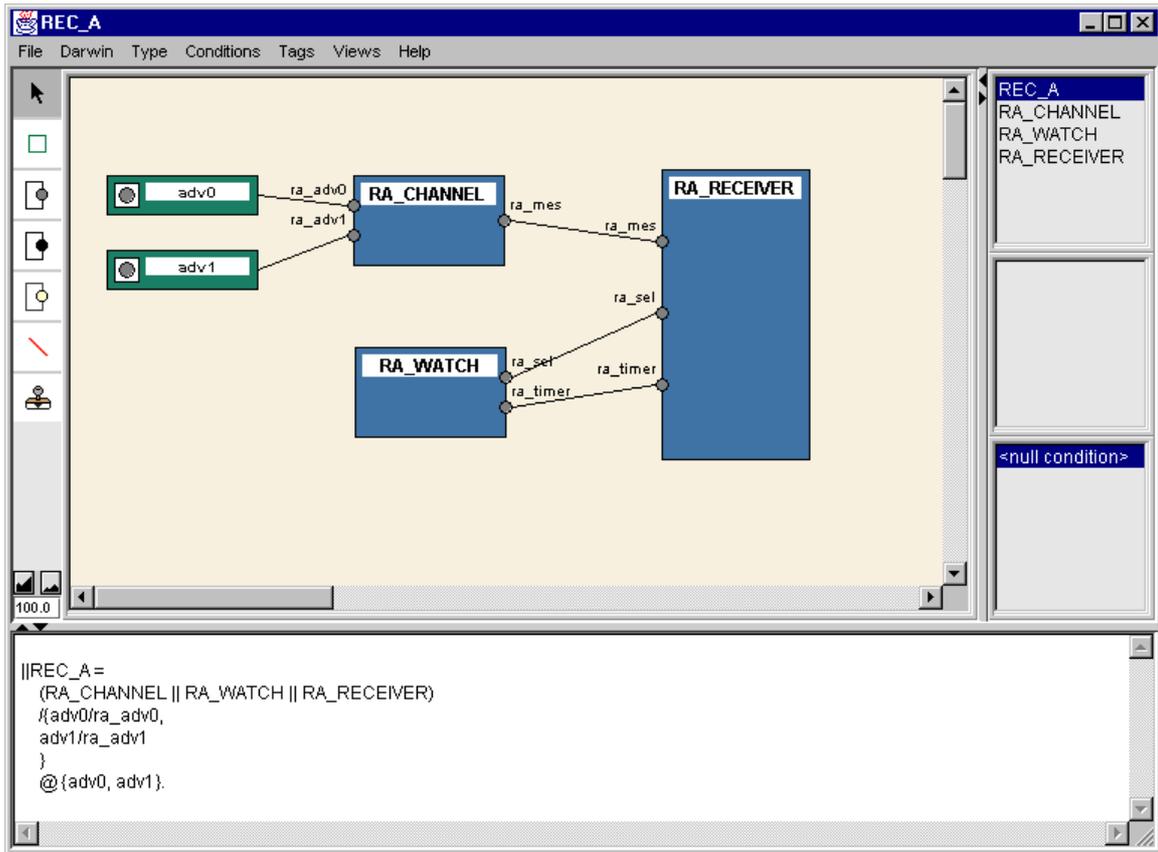


Figure 11. REC\_A in Darwin using the SAA

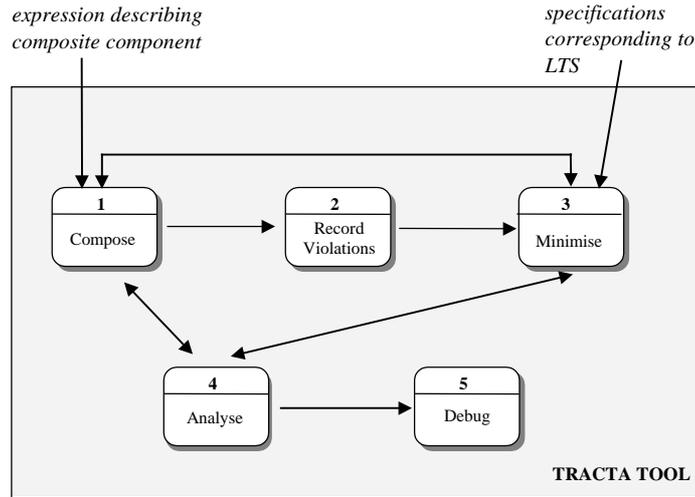
## 4.2 Tool implementation

In this section, we describe the structure of the Tracta tool for analysing the behaviour of distributed systems. As the tool is to be replaced by the LTSA tool, we concentrate on the algorithms that have been used to implement its main functionality. These are independent of the specific implementation. Moreover, they form the basis of the new tool.

The modules of the tool that implement the Tracta approach are illustrated in Figure 12, where arrows describe the flow of information between modules.

**Module 1 – Compose.** This module computes the LTS for a composite component from that of its sub-components. The LTS returned can be analysed or further composed. A standard composition algorithm can be found in (Holzmann 91).

Among the LTSs composed, some may express safety and liveness properties or user-specified interfaces (Cheung and Kramer 96b). Module “Compose” automatically transforms these as required by Tracta, before performing the composition. As mentioned, the globally unique action  $@name$  labels accepting transitions of a Büchi automaton, where  $name$  is the identifier of the corresponding liveness property. This simplifies the identification of properties when violations are detected. Finally, state  $\pi$  is internally represented by the value -1.



**Figure 12. Modules of the Tracta tool**

**Module 2 - Record Violations.** When fair choice is not a desirable assumption in the model of a system, the intermediate graphs obtained by compositional analysis need to be modified according to the technique presented in (Giannakopoulou 98a). The technique relies on the computation of  $\tau$ -strongly connected components in a state-graph  $G$ . The module uses the algorithm proposed by Aho et al. (Aho, et al. 74) to compute the strongly connected components in the sub-graph of  $G$  defined by the  $\tau$ -relation. The complexity of the algorithm is linear in the size of the graph.

**Module 3 - Minimise.** This module minimises a given LTS  $L$  with respect to Milner’s weak semantic equivalence (Milner 89). Due to the fact that minimisation takes up most of the computational effort in our analysis method, the module implements the algorithm presented by Fernandez (Fernandez 90). In general, strong equivalence (Milner 89) can be tested in  $O(mn)$  time for a labelled transition system with  $m$  transitions and  $n$  states (Kanellakis and Smolka 90). However, the problem can be reduced to the relational coarsest partition problem, which has been solved by Paige and Tarjan (Paige and Tarjan 87) in  $O(m \log n)$  time. The algorithm proposed by Paige and Tarjan has been adapted by Fernandez to minimise labelled transition systems modulo strong equivalence. Minimising  $P = \langle S, A, \Delta, q_0 \rangle$  with respect to weak equivalence, is reducible to minimising  $P' = \langle S, A, \Delta', q_0 \rangle$  with respect to strong equivalence, where  $\Delta'$  is obtained from  $\Delta$  by allowing  $\tau$  actions to be absorbed into visible actions (Kanellakis and Smolka 90). Our algorithm creates two initial partitions for an LTS  $L$ , one containing the  $\pi$  state, and one containing the remaining states of  $L$ . It then proceeds as prescribed in (Fernandez 90), by refining these partitions until stability is reached. The  $\pi$  state has to form a partition by itself since it is by definition not equivalent to any other state (section 2.2).

**Module 4 – Analyse.** This module collects information obtained during stages 1 and 3 and additionally uses the LTSs that these modules generate in order to report the following:

- timing results for the algorithms of composition and minimisation.
- sizes of graphs (in terms of numbers of states and transitions) for the subsystem after composition and after minimisation.
- detection of safety or liveness property violations. When a liveness property violation is detected in an LTS, the identity of the root of a terminal set of states is returned, together with the actions labelling transitions between states in the terminal set of states. Violation of safety properties is detected by the existence of state  $\pi$  in the global system graph.
- detection of deadlock states in the composite LTS. Deadlocks are detected on the system graph as states that have no outgoing transitions.

The report obtained by this module may be used by the developer in combination with module “Debug”, in order to obtain useful diagnostic information that will guide the correction of errors in the design.

**Module 5 - Debug:** Given an LTS  $L$  and a state  $s$  in  $L$ , this module produces a trace of  $L$  leading to  $s$ . Breadth-first traversal of the graph guarantees the fact that the trace obtained is the shortest possible.

For the case of liveness properties, the module is used to obtain the shortest possible trace to the root of a violating terminal set of states.

For the case of safety properties, the module is used to obtain a trace to the violating state  $\pi$ . This trace provides a scenario that demonstrates how a safety property can be violated. As mentioned in section 3.2, when multiple safety properties are simultaneously analysed, the tool can detect safety violations but it cannot distinguish which property is violated. If the trace does not identify a property, then it must be used to trace the violation down the compositional hierarchy.

### 4.3 Checking the RMTP protocol: experimental results

In our case study of the RMTP protocol, we aim at an abstracted view of the system where the effect of failures of designated receivers on the stability of REC\_B can be observed. As described in section 2.3, REC\_B is considered stable whenever it has a currently selected acknowledgement processor. We have decided to check the stability of REC\_B, as it is the ordinary receiver at the lowest level of the multicast tree, and is therefore affected by the failures of all designated receivers in the tree. Consequently with the exception of actions *dra\_fail*, *drb\_fail*, and *rb\_stable*, all other actions are hidden as soon as they are considered internal to a subsystem.

As discussed in section 3.2, we have included safety property SAF\_LC\_REG1 in our case study. This property refers to subsystem LOC\_REG1 (see Figure 4). As the  $\pi$  state is reachable in LOC\_REG1, we conclude that property SAF\_LC\_REG1 is violated by the subsystem. This violation is not remedied in the system, as the  $\pi$  state is also reachable in AP\_SELECT.

The tool returns the following trace of LOC\_REG1, which leads to the violation of property SAF\_LC\_REG1:  $\langle adv.2, adv.2, drb\_fail, rb\_sel.2, rb\_sel.2 \rangle$ . This trace shows that the violation may occur when DES\_REC\_B, prior to its failure, broadcasts two advertisements along its sub-tree. Since *rb\_sel.2* can occur twice, we conclude that RB\_CHANNEL has transmitted both advertisements to RB\_RECEIVER. The trace helps us construct the following scenario that shows in detail how the property can be violated. RB\_CHANNEL receives an advertisement from DES\_REC\_B, transmits it to RB\_RECEIVER, and then receives another advertisement from DES\_REC\_B. RB\_RECEIVER selects DES\_REC\_B as its acknowledgement processor. Subsequently a time-out occurs, which initiates the selection mechanism of RB\_RECEIVER. RB\_CHANNEL still contains the second advertisement, which it now transmits to RB\_RECEIVER. As a result, RB\_RECEIVER performs the second *rb\_sel.2*.

This violation represents a typical situation in distributed systems. In an asynchronous environment, channels need to be equipped with substantial buffers. Consequently, nodes of the system may be receiving old messages from a failed node, thus getting the impression that the node is still alive. In such situations, property SAF\_LC\_REG1 would be turned into a property, which states that: “at some stage subsequent to the failure of DES\_REC\_B, REC\_B never again selects DES\_REC\_B as its acknowledgement processor”. In our simplified case study where the channel has capacity one, it is enough for the property to allow the occurrence of *rb\_sel.2* at most twice subsequent to *drb\_fail*. This modification to property SAF\_LC\_REG1 removes the violation from the system analysed.

Liveness properties are checked on the global graph for the system. By analysing AP\_SELECT we have found that property LV\_RECA is satisfied, as transition  $@LV\_RECA$  is enabled in all terminal sets of states of AP\_SELECT. On the other hand, property LV\_RECB is violated, and the tool returns trace  $\langle drb\_fail, dra\_fail \rangle$ . The trace provides a path to the root of a terminal set of states where  $@LV\_RECB$  is not enabled. However, this represents legal behaviour of the system, as DES\_REC\_B may fail very early in a multicast session, before any of its advertisements are received by REC\_B in its sub-tree. In order to verify that there is no error in our design, we disable action *drb\_fail* and analyse the property again. Indeed when DES\_REC\_B never fails, REC\_B eventually selects DES\_REC\_B as its acknowledgement processor.

After analysing the properties introduced in our case study, accepting transitions are removed, and the resulting LTS is minimised. The abstracted behaviour of AP\_SELECT that we aimed at is thus obtained, and is illustrated in

Figure 13. It is clear from this view of the system that failures of designated receivers do not affect the stability of REC\_B.

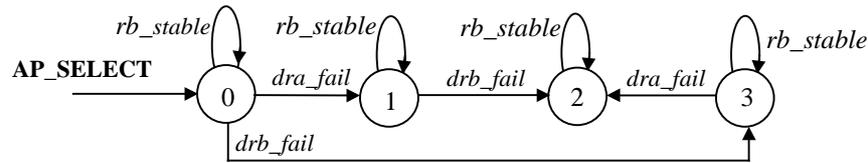


Figure 13. Final LTS obtained for AP\_SELECT

The Tracta tool has been used for performing the RMTP case study using standard compositional reachability analysis (CRA) and traditional reachability analysis (RA). In CRA, analysis proceeds in the same way as in Tracta. The difference is that property automata are not included in the compositional hierarchy. As a result, actions involved in liveness properties need to be globally exposed for these properties to be checked. To be fair in our comparisons, we have not exposed internal actions involved in the safety property when performing CRA. As already mentioned, the safety property of our RMTP case study can be checked at a subsystem level.

| RMTP – Dynamic<br>Selection of AP              | Tracta  |         | CRA                 |                        | Traditional RA            |                             |
|--|---------|---------|---------------------|------------------------|---------------------------|-----------------------------|
|  | #states | #trans. | #states             | #trans.                | #states                   | #trans.                     |
| Largest subsystem<br>(before minimisation)     | 153     | 1096    | 252                 | 1286                   | not applicable            |                             |
| System to be analysed<br>(before minimisation) | 129     | 869     | 236<br>(properties) | 1151<br>(not included) | 720 000<br>(properties)   | 5 724 208<br>(not included) |
| System to be analysed<br>(after minimisation)  | 46      | 254     | 186<br>(properties) | 863<br>(not included)  | not applicable            |                             |
| Final system<br>(after analysis)               | 4       | 8       | 4                   | 8                      | 720 000<br>(too large to) | 5 724 208<br>(minimise)     |

Table 1. Size of RMTP case study for different approaches to analysis

In Table 1, we compare Tracta and CRA in terms of the size of the LTSs obtained. Both in TRACTA and CRA, it is possible to obtain an intermediate subsystem that is larger than the final system. However the sizes of these subsystems are small enough so as not to require the inclusion of contextual interfaces into analysis (Cheung and Kramer 96b). When compared to CRA, Tracta achieves a reduction in the size of LTSs at all levels of the hierarchy, even though in Tracta property LTSs have been included in the system.

Traditional reachability analysis computes the reachable states of a system from the LTSs of its components, in a single step. From Table 1, it is obvious that the RMTP example justifies a compositional approach to analysis. With traditional reachability analysis, a system of 720 000 states and 5 724 208 transitions needs to be analysed, as compared to a system of 46 states and 254 transitions in Tracta. The system obtained by traditional reachability analysis is too large to be minimised by our tool. We are thus unable to generate an abstraction of the system behaviour, which we have done with Tracta and CRA.

The considerable reduction achieved by compositional methods in this case study stems from the fact that a large amount of interleaving is involved between actions that may be made internal to subsystems. Traditional reachability analysis computes all possible interleaving. Compositional approaches simplify the interleaving between internal actions at intermediate stages of the analysis.

## 5. Conclusions and future work

Analysis plays a significant role in the design of complex systems. Software developers are encouraged to identify critical and error-prone parts in a system behaviour by the construction of models. Analysis then provides the capability of automatically checking these models thereby increasing confidence in a design. In order to be more readily usable, analysis must form an integral part of the software development process. In this paper, we have presented the way in which this goal has motivated our choices when developing the Tracta approach - a compositional technique for analysing the behaviour of distributed systems based on their software architecture.

In our approach, software architectures described in Darwin provide the common structure or skeleton of the system during development, i.e. behaviour specification, behaviour analysis, and system implementation. The architectural hierarchy of components in the system, as described in Darwin, is used to direct the Tracta approach into performing analysis compositionally. This is one of the main advantages of Tracta as compared to other CRA techniques. With Tracta, analysis is not an isolated stage in the software development process. Rather, there is a flow of information between the model for analysis and the design for construction. This information includes changes imposed to the basic structure of the system during development.

One of the appealing characteristics of Tracta is the uniformity of the formalisms for behaviour and property specification. Both properties and processes in the system are specified in terms of state machines. Checking is performed by inclusion of properties in the compositional hierarchy. It is a unique feature of Tracta, and one of its main contributions in the domain of CRA techniques, that its property checking mechanisms are independent of actions hidden at intermediate phases of analysis. Moreover, properties introduced into analysis may play a significant role in reducing the size of the state space to be analysed in the presence of illegal behaviours. All transformations introduced by the Tracta checking mechanisms are performed automatically and transparently to the user. Finally, besides returning useful results about the system under analysis, Tracta generates an abstracted view of system behaviour, according to the user requirements.

Throughout the paper, we have used the case study of a reliable multicast transport protocol (RMTP) to illustrate our discussions. The case study has demonstrated that compositional approaches may significantly reduce state explosion. Moreover, the property checking mechanisms introduced by Tracta may further increase the reduction obtained with standard CRA techniques.

Although the Tracta approach may significantly reduce the size of the system to be analysed, it is still susceptible to state explosion. In fact, compositional analysis does not always prove more efficient than reachability analysis, especially when the latter is performed “on the fly” and is combined with partial-order reduction (Fernandez, et al. 96, Holzmann, et al. 92). Our future plans include comparing Tracta with such techniques in order to identify, if possible, the kinds of systems for which one or the other approach proves more efficient.

A further extension to the Tracta tool that is under consideration is to provide automatic support into translating LTL formulae into Büchi automata for analysis (Gerth, et al. 95). However, such algorithms do not construct minimal automata for the properties. In our experience, the liveness properties commonly checked tend to fall into some basic patterns. Therefore, rather than algorithmically deriving the property automata, we are experimenting with the provision of basic templates for automata corresponding to those patterns most frequently encountered. As well as providing guidance in the expression of properties, Tracta should also ideally assist the user in locating these properties into the compositional hierarchy of the system.

An issue that arises in all compositional reachability analysis approaches is that diagnostic information obtained at the global system graph may not contain enough information to be of practical use. An advantage of the Tracta safety checking mechanism is that violations are recorded at intermediate systems with state  $\pi$ . As a result, a counterexample obtained at the global level may be enriched with details obtained by tracing the violation down the compositional hierarchy. We intend to provide further assistance into this process, as well as in the process of identifying the safety property violations that correspond to the existence of state  $\pi$  in the final system.

Finally we are working on completing the integration of the LTSA tool supporting Tracta with the SAA tool, which provides the visual environment for the design and construction of distributed systems. Although further work is certainly necessary in all of the above aspects, the basic framework for performing compositional analysis by exploiting software architectures has already been provided in the Tracta approach, and offers a promising ground for experimentation and future enhancements.

## Acknowledgements

We gratefully acknowledge helpful discussions with our colleague, Jeff Magee, and the financial support provided by the following grants: the EPSRC Grant GR/J 87022 (TRACTA Project), the EU (ARES Framework IV contract 20477), the RGC grant HKUST 662/95E, and the British Council UK/HK Joint Research Scheme project JRS96/38. We also thank the anonymous referees for their valuable comments.

## References

- Aggarwal, S., Courcoubetis, C., and Wolper, P., 90. "Adding Liveness Properties to Coupled Finite-State Machines," *ACM Transactions on Programming Languages and Systems*, vol. 12(2), pp. 303-339, April 1990.
- Aho, A.V., Hopcroft, J.E., and J.D.Ullman, 74. *The Design and Analysis of Computer Algorithms*: Addison-Wesley.
- Andrews, G.R., 91. *Concurrent Programming - Principles and Practice*: The Benjamin / Cummings Publishing Company Ltd.
- Cheung, S.C., Giannakopoulou, D., and Kramer, J., 97. "Verification of Liveness Properties using Compositional Reachability Analysis," in *Proc. of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'97)*, Zurich, Switzerland, September 1997. Lecture Notes in Computer Science 1301, pp. 227-243. M. Jazayeri and H. Schauer, Eds.
- Cheung, S.C. and Kramer, J., 96a. "Checking Subsystem Safety Properties in Compositional Reachability Analysis," in *Proc. of the 18th International Conference on Software Engineering*, Berlin, Germany, March 1996, pp. 144-154.
- Cheung, S.C. and Kramer, J., 96b. "Context Constraints for Compositional Reachability Analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 5(4), pp. 334-377, October 1996.
- Clarke, E.M. and Wing, J.M., 96a. "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys*, vol. 28(4), pp. 626-643.
- Cleaveland, R., Parrow, J., and Steffen, B., 93b. "The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems," *ACM Transactions on Programming Languages and Systems*, vol. 15(1), pp. 36-72, January 1993.
- Fernandez, J.-C., 90. "An Implementation of an Efficient Algorithm for Bisimulation Equivalence," *Science of Computer Programming*, vol. 13(2-3), pp. 219-236, May 1990.
- Fernandez, J.-C., Mounier, L., Jard, C., and Jéron, T., 92. "On-the-fly Verification of Finite Transition Systems," *Formal Methods in System Design*, vol. 1(2/3), pp. 251-273, October 1992.
- Fernandez, J.-C., Garavel, H., Kerbrat, A., Mateescu, R., Mounier, L., and Sighireanu, M., 96. "CADP: A Protocol Validation and Verification Toolbox," in *Proc. of the 8th International Conference on Computer-Aided Verification (CAV'96)*, New Brunswick, NJ, USA, July/August 1996. Lecture Notes in Computer Science 1102, pp. 437-440. R. Alur and T. A. Henzinger, Eds.
- Gerth, R., Peled, D., Vardi, M.Y., and Wolper, P., 95. "Simple On-the-fly Automatic Verification of Linear Temporal Logic," in *Proc. of the 15th IFIP/WG6.1 Symposium on Protocol Specification, Testing and Verification (PSTV'95)*, Warsaw, Poland, June 1995, pp. 3-18.
- Ghezzi, C., Jazayeri, M., and Mandrioli, D., 91. *Fundamentals of Software Engineering, Chapter 6*: Prentice-Hall International.
- Giannakopoulou, D., 95. "The TRACTA Approach for Behaviour Analysis of Concurrent Systems," Dept. of Computing, Imperial College, London, Research Report DoC 95/16, September 1995.
- Giannakopoulou, D., 98a. "Model Checking for Concurrent Software Architectures," PhD thesis, Dept. of Computing, Imperial College, London. *In Preparation*.

- Godefroid, P. and Holzmann, G.J., 93. "On the Verification of Temporal Properties," in *Proc. of the 13th IFIP WG 6.1 International Symposium, on Protocol Specification, Testing, and Verification (PSTV'93)*, Liège, Belgium, June 1993, pp. 109-124. A. Danthine, G. Leduc, and P. Wolper, Eds.
- Godefroid, P. and Wolper, P., 91. "Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties," in *Proc. of the 3rd International Workshop on Computer Aided Verification (CAV'91)*, Aalborg, Denmark, July 1991. Lecture Notes in Computer Science 575, pp. 332-342. K. G. Larsen and A. Skou, Eds.
- Graf, S. and Steffen, B., 90. "Compositional Minimization of Finite State Systems," in *Proc. of the 2nd International Conference on Computer-Aided Verification (CAV'90)*, New Brunswick, NJ, USA, June 1990. Lecture Notes in Computer Science 531, pp. 186-196. E. M. Clarke and R. P. Kurshan, Eds.
- Gribomont, P. and Wolper, P., 89. "Temporal Logic," in *From Modal Logic to Deductive Databases*, A. Thayse, Ed.: John Wiley and Sons.
- Hoare, C.A.R., 85. *Communicating Sequential Processes*: Prentice-Hall.
- Holzmann, G.J., 91. *Design and Validation of Computer Protocols*: Prentice Hall.
- Holzmann, G.J., Godefroid, P., and Pirottin, D., 92. "Coverage Preserving Reduction Strategies for Reachability Analysis," in *Proc. of the IFIP/WG6.1 International Symposium on Protocol Specification, Testing, and Verification (PSTV'92)*, Orlando, Florida, June 1992, pp. 349-364.
- Kanellakis, P.C. and Smolka, S.A., 90. "CCS Expressions, Finite State Processes, and Three Problems of Equivalence," *Information and Computation*, vol. 86(1), pp. 43-68, May 1990.
- Kemppainen, J., Levanto, M., Valmari, A., and Clegg, M., 92. "'ARA' Puts Advanced Reachability Analysis Techniques Together," in *Proc. of the 5th Nordic Workshop on Programming Environment Research*, Tampere, Finland, January 1992, pp. 233-257. K. Systä, P. Kellomäki, and R. Mäkinen, Eds.
- Kramer, J. and Magee, J., 97. "Exposing the Skeleton in the Coordination Closet," in *Proc. of the Coordination'97, Second International Conference on Coordination Models and Languages*, Berlin, Germany, September 1997. Lecture Notes in Computer Science 1282, pp. 18-31. D. Garlan and D. I. Métayer, Eds.
- Krimm, J.-P. and Mounier, L., 97. "Compositional State Space Generation from Lotos Programs," in *Proc. of the 3d International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, Enschede, The Netherlands, April 1997. Lecture Notes in Computer Science 1217. E. Brinksma, Ed.
- Lin, J.C. and Paul, S., 96. "RMTP: A Reliable Multicast Transport Protocol," in *Proc. of the IEEE INFOCOMM'96*, San Francisco, California, March 1996, pp. 1414-1424.
- Magee, J., Dulay, N., Eisenbach, S., and Kramer, J., 95. "Specifying Distributed Software Architecture," in *Proc. of the 5th European Software Engineering Conference (ESEC'95)*, Sitges, Spain, September 1995. Lecture Notes in Computer Science 989, pp. 137-153. W. Schäfer and P. Botella, Eds.
- Magee, J., Dulay, N., and Kramer, J., 94. "A Constructive Development Environment for Parallel and Distributed Programs," in *Proc. of the Second International Workshop on Configurable Distributed Systems*, Pittsburg, Pennsylvania, March 1994, pp. 4-14.
- Magee, J., Kramer, J., and Giannakopoulou, D., 97. "Analysing the Behaviour of Distributed Software Architectures: a Case Study," in *Proc. of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*, Tunis, Tunisia, October 1997, pp. 240-245.
- Manna, Z. and Pnueli, A., 92. *The Temporal Logic of Reactive and Concurrent Systems - Specification*: Springer-Verlag.
- McMillan, K.L., 93. *Symbolic Model Checking*: Kluwer Academic Publishers.
- Milner, R., 89. *Communication and Concurrency*: Prentice-Hall.
- Ng, K., Kramer, J., Magee, J., and Dulay, N., 96. "A Visual Approach to Distributed Programming," *Tools and Environments for Parallel and Distributed Systems*, pp. 7-31, February 1996.

- Paige, R. and Tarjan, R.E., 87. "Three Partition Refinement Algorithms," *SIAM Journal of Computing*, vol. 16(6), pp. 973-989, 1987.
- Rabinovich, A., 92. "Checking Equivalences Between Concurrent Systems of Finite Agents," in *Proc. of the 19th International Colloquium on Automata, Languages and Programming*, Wien, Austria, July 1992. Lecture Notes in Computer Science 623, pp. 696-707. W. Kuich, Ed.
- Roscoe, A.W., 94. "Model-checking CSP," in *A Classical Mind: Essays in Honour of C.A.R. Hoare*, Prentice Hall International Series in Computer Science, A. W. Roscoe, Ed.: Prentice-Hall, pp. 353-378.
- Sabnani, K.K., Lapone, A.M., and Uyar, M.Ü., 89. "An Algorithmic Procedure for Checking Safety Properties of Protocols," *IEEE Transactions on Communications*, vol. 37(9), pp. 940-948, September 1989.
- Tai, K.C. and Koppol, P.V., 93. "An Incremental Approach to Reachability Analysis of Distributed Programs," in *Proc. of the 7th International Workshop on Software Specification and Design*, Los Angeles, California, December 1993.
- Tarjan, R., 72. "Depth-First Search and Linear Graph Algorithms," *SIAM Journal of Computing*, vol. 1, pp. 146-160, 1972.
- Valmari, A., 92. "Alleviating State Explosion during Verification of Behavioural Equivalence," Department of Computer Science, University of Helsinki, Finland, Research Report A-1992, August 1992.
- Valmari, A., 93b. "Compositional State Space Generation," in *Proc. of the Advances in Petri Nets*, Leiden, The Netherlands, 1993. Lecture Notes in Computer Science 674, pp. 427-457. G. Rozenberg, Ed.
- Vardi, M.Y. and Wolper, P., 86. "An automata-theoretic approach to automatic program verification," in *Proc. of the 1st Symposium on Logic in Computer Science*, Cambridge, June 1986, pp. 322-331.
- Yeh, W.J., 93a. "Controlling State Explosion in Reachability Analysis," SERC, Purdue University, PhD Thesis SERC-TR-147-P, December 1993.