TRACTA: An Environment for Analysing the Behaviour of Distributed Systems

Dimitra Giannakopoulou, Jeff Kramer  Shing Chi Cheung
Department of Computing  Department of Computer Science
Imperial College of Science, Technology and Medicine, London SW7 2BZ, UK  Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong
Email: {dgl, jk}@doc.ic.ac.uk  Email: scc@cs.ust.hk

ABSTRACT

Particular emphasis needs to be placed on the integration of analysis techniques with other software development activities, to form a complete environment for the design and construction of distributed systems. We have addressed this problem by using 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 to incrementally construct 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 also composed into the system and violations are detected on the global graph obtained. The method is supported by an automated tool implemented in C++ and running on Unix.

1. INTRODUCTION

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 program performs as intended.

Reachability analysis of a finite-state model can be used to prove that a program behaves as specified. This type of analysis consists of constructing the product state-space of all components in the system, with each finite-state component modelling the behaviour of a synchronously interacting process. Various approaches to definition of system specifications are used. Often the system is specified as a finite-state machine, in which case the product state-space is checked for equivalence with its specifications. A more common approach, however, is to provide specifications in terms of a set of properties that the system needs to satisfy. Such properties can be expressed as formulas in some temporal or modal logic [1, 2] or as property automata [3-6], against which the model of the system is verified.

Analysis is not practical unless supported by some automated tools. Reachability analysis therefore owes much of its popularity to the fact that it is fairly easy to automate. The main disadvantage of this technique is the state explosion problem which can occur if the system being verified has many components that can make transitions in parallel. Great effort has been made to avoid this problem by not having to construct the complete graph. Automated methods that exhaustively search the problem state space can be roughly classified into two categories: reduction by partial orders avoids the generation of all paths formed by the interleaving of the same set of transitions [7, 8]; reduction by compositional minimisation bases reduction on intermediate simplification of subsystems [9, 10]. Techniques in the latter category are known as compositional reachability analysis (CRA).
In CRA, analysis is performed in two steps. Firstly, 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. The key to the success of such techniques is to hide as many internal actions as
possible in each subsystem. The observable behaviour of a subsystem where internal details have been
hidden can generally be represented by a smaller state machine. The approach is particularly suitable for
analysing programs which are likely to evolve, as the effect of change is localised. When changes are
applied to a program, only state machines of those subsystems that are affected by the changes need be
re-computed.

Tracta is a CRA method supported by an automated tool. In the method, properties are expressed as
automata which are composed into the system global-graph. These properties are transparent to the
system unless they are violated, in which case they are detected. Tracta has the power to validate
multiple properties at the same time. Moreover the user is free to abstract from internal details of
subsystems without compromising the effectiveness of the analysis.

The Tracta CRA approach has been specifically designed to suit the hierarchical composition of
components which is used in the development of distributed systems [11]. The compositional approach
therefore not only helps reduce the state explosion problem, but also makes Tracta compatible with the
structural way of building distributed software. Further, it supports flexibility in validating properties
irrespective of details hidden at various stages of the design.

Various approaches have been proposed for the static analysis of concurrent systems, a number of them
supported by automated tools. In the Concurrency Factory [1] process algebra is used as the underlying
formal model of computation. The main verification tool is a linear-time model checker for the
alternation-free modal μ-calculus. Aldebaran [12] is a verification tool for communicating processes
expressed as labelled transition systems (LTS). It permits reduction and comparison of LTS with respect
to some equivalence of interest. SPIN [13] performs reachability analysis on a set of processes specified
as labelled transition systems in the specification language PROMELA. Correctness criteria that can be
expressed in PROMELA define behaviours that are claimed to be impossible. The Symbolic Model
Verifier system [2] performs symbolic model checking, where the program state space is represented
symbolically rather than explicitly. Properties are described in the temporal logic Computation Tree
Logic (CTL). The distributed PAL system [14] is a CRA tool which uses process algebra for specifying
interacting processes. Verification is performed by equivalence checking of the global-system graph with
the system specifications. With the exception of the PAL system, all of the above tools perform
reachability analysis in a single step. However, validation of specifications in PAL is restricted to those
involving actions that remain globally observable.

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 its behaviour. The remainder of the paper is organised as follows: section 2 is a brief presentation of
labelled transitions and of the method that we use for proving properties of distributed systems. Section
3 illustrates the concepts presented in section 2 by means of the case study of a reliable multicast
transport protocol. We proceed by introducing the environment for distributed system design with which
Tracta is designed to interact, as well as the current version of the Tracta tool, in section 4. We conclude
with discussion and plans for future work.

2. TRACTA AS A METHOD

Labelled transition systems (LTS) can be used to model the behaviour of synchronous communicating
processes in a distributed program. An LTS contains all the states a process may reach and all the
transitions it may perform. The model has been widely used in the literature for specifying and analysing
distributed programs [15-19]. Formally, an LTS of a process P is a quadruple \(< S, A, Δ, q >\) where
(i) \( S \) is a set of states;

(ii) \( A = \alpha P \cup \{\tau\} \), where \( \alpha P \) denotes the communicating alphabet of \( P \) which does not contain the internal action \( \tau \);

(iii) \( \Delta \subseteq S \times A \times S \), denotes a transition relation that maps from a state and an action onto another state;

(iv) \( q \) is a state in \( S \) which indicates the initial state of \( P \).

An LTS of \( P = < S, A, \Delta, q > \) transits into another LTS of \( P' = < S, A, \Delta, q' > \) with an action \( a \in A \) if and only if \((q, a, q') \in \Delta \). That is,

\[
< S, A, \Delta, q > \not\equiv < S, A, \Delta, q' > \iff (q, a, q') \in \Delta.
\]

Since there is a one-to-one mapping between a process \( P \) and its LTS, we use the terms process and LTS interchangeably.

A set of states \( C \) in an LTS \( < S, A, \Delta, q > \) 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.,

\[
\forall s \in C, (s, a, s') \in \Delta \Rightarrow s' \in C.
\]

Observability of actions in a process can be controlled by a restriction operator \( \uparrow \). \( P \uparrow L \) represents the process projected from \( P \) in which actions in \( \alpha P \)-\( L \) are replaced by the internal action \( \tau \). Concurrent LTSs are synchronised via transitions labelled with the same communicating actions. The parallel composition operator \( \parallel \) combines the behaviour of two LTS \( P \) and \( Q \), into that of another LTS \( R = P \parallel Q \). Every state of \( R \) reflects the respective states of \( P \) and \( Q \). \( R \) is the LTS that results by synchronisation of actions common to the alphabets of \( P \) and \( Q \), and by interleaving of their individual actions. The composition operator is commutative and associative.

The software architecture of a distributed program is represented as a hierarchical composition of subsystems, with interacting primitive components at the leaves of the hierarchy, as described in section 4.1. Each primitive component is modelled as an LTS. The LTS of the overall system is then composed stepwise from those of its subsystems in a bottom-up manner. At each intermediate step, the LTS of a subsystem is simplified by hiding internal actions that are of no interest to the global view of the system. Such abstraction of internal details is achieved with the restriction operator \( \uparrow \). After every composition step, minimisation of the resulting LTS is performed with respect to weak equivalence as defined by Milner [20]. Weak equivalence equates processes that depict identical behaviour to an external observer. The observable behaviour of a subsystem where internal actions have been hidden can generally be represented by a simpler LTS.

A system expressed in this way can then be checked against its specifications. Specifications are expressed as a set of properties that the system needs to satisfy. Properties are separated into two classes: safety and liveness. A safety property asserts that the program never enters an undesirable state [21]. For example, mutual exclusion is a safety property which specifies the absence of a program state where a common resource is simultaneously accessed by more than one client. A liveness property asserts that a program eventually enters a desirable state [21]. For example, the assertion that a program will eventually close a file after opening it is a liveness property.

A safety property is specified by the user as an LTS \( P = < S, A, \Delta, q > \). Such an LTS specifies subsystem behaviour as related to a set of actions of interest. For verification reasons, \( P \) is converted to its image
process \( P' = \langle S \cup \{\pi\}, A, \Delta', q \rangle \), where \( \Delta' \) is constructed from \( \Delta \) by the following procedure:

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

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

\( P' \) is inserted in the compositional hierarchy of the system, to be composed with the system to which it refers. The global graph of the system is then checked for the existence of state \( \pi \), which reflects the violation of property \( P \) by the system under analysis.

A liveness property is expressed as a Büchi automaton \( B = \langle S, A, \Delta, q_0, F \rangle \), where \( S \) is a finite set of states, \( A \) is a set of observable actions, \( \Delta \) is a set of transitions, \( q_0 \) is its initial state, and \( F \) is a set of acceptance states. An infinite word \( a_1a_2a_3\ldots \) over \( A \) is accepted by \( B \) if and only if there exists an infinite execution \( q_0 \overset{a_1}{\rightarrow} q_1 \overset{a_2}{\rightarrow} q_2 \overset{a_3}{\rightarrow} \ldots \) of \( B \) such that \( q_i \in F \) for infinitely many \( i \)'s. \( B \) is mapped to a liveness property LTS \( B' = \langle S, A \cup \{acc\}, \Delta', q \rangle \), by adding a new globally unique action \( acc \) and new transitions such that:

(i) \( acc \in aA; \) and

(ii) \( \Delta' = \Delta \cup \{ s \overset{acc}{\rightarrow} s | s \in F \} \).

Similarly to the case for safety properties, \( B' \) is inserted in the compositional hierarchy of the system. Satisfaction of \( B \) is equivalent to appearance of acceptance transitions (transitions labelled with \( acc \)) at all terminal sets of states in the global graph for the system \(^2\). Our approach to verification of safety and liveness properties is described in detail in [4] and [3] respectively.

As mentioned, 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. However, the properties that are available for reasoning in the analysis are then constrained by the remaining globally observable actions. An advantage that Tracta presents over other compositional methods is that it makes the verification of properties independent from the actions that are globally observable. The users are therefore left free to choose the view of the system in which they are interested, by hiding actions at various phases of the analysis without compromising its effectiveness.

3. **MODELLING THE RMTP PROTOCOL**

To illustrate our approach, we introduce a Reliable Multicast Transport Protocol (RMTP) as proposed by Lin and Paul [22]. The protocol is designed for applications that cannot tolerate data loss. It provides sequenced, lossless delivery of data from a sender to a group of receivers, at the expense of delay. Reliability is achieved by a periodic transmission of status by the receivers (ACK packets) and a selective retransmission mechanism by the sender. Scalability is provided by grouping receivers into a hierarchy of local regions, with a Designated Receiver (DR) in each of those. Receivers in each local region send their ACKs to the corresponding DR, DRs send their ACKs to the higher level DRs or to the sender (see Figure 1), thereby avoiding the ACK-implosion problem. In addition, DRs cache received data and respond to receivers in their local regions, thus decreasing end-to-end latency. The term Acknowledgement Processor (AP) is used to denote either a DR or the sender, when referring to them as entities that receive and process ACKs.

---

\(^1\) The state \( \pi \) is an artificial state, introduced as a trap state into analysis, for capturing traces of the system behaviour that violate its specifications.

\(^2\) This holds under the assumption of fair selection and fair process execution in the modelled systems. The method can be easily adapted to handle cases where the assumption cannot be made. However, this is beyond the scope of this paper.
To cater for situations where DRs 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 we focus on. Dynamic selection of APs is achieved in RMTP by the use of a special packet, called the SND_ACK_TOME (SAT) packet. The sender and all DRs periodically advertise themselves (action $adv$) by multicasting SAT packets along their subtrees. All SAT packets have the same initial time-to-live (TTL) values in the header. Routers decrement the TTL value when forwarding packets. Therefore a larger TTL is equivalent to a closer proximity in the multicast tree. A receiver stores the address and TTL value of its selected AP, and selects a new one as soon as a SAT packet with a higher TTL is received. To recover from the failure of the selected AP, a time-out is also used to initiate the selection mechanism.

The configuration used in our case study, is depicted in Figure 1. Three processes are associated with every simple receiver in the multicast tree. For example, the behaviour of REC_1 in Figure 1 is given by the composite behaviour of the Receiv_1, Chnl_1 and Watch_1 LTS processes, specified as in Figure 2.

![Figure 1: A Multicast Tree of Receivers](image-url)
The Chnl_1 process models a lossy channel which receives advertisements from the APs above the receiver (actions \textit{advA/B/S}) and transmits them to the Receiv_1 process (actions \textit{mesA/B/S}), or loses them (action \textit{lose}). The specification assumes fair execution in the sense that unfair execution sequences where Chnl_1 keeps losing all messages are refused. The Watch_1 process models the time-out associated with the selection of a new AP. It observes all potential APs for the receiver, and when a failure of the selected AP occurs (actions \textit{failA/B}), it informs Receiv_1 (actions \textit{ms failA/B}) so that the selection procedure is initiated. Receiv_1 then selects as its AP (action \textit{selA/B/S}) the AP whose advertisement it receives first. Selections are modified whenever an advertisement is received from a
nearer AP than the one currently selected. In the composite behaviour of the Receiv_1, Chnl_1 and Watch_1 processes, only actions failA/B, and advA/B/S synchronise with the environment of REC_1, so all the remaining actions are made unobservable, i.e.,

\[ \text{REC}_1 = (\text{Chnl}_1 \parallel \text{Watch}_1 \parallel \text{Receive}_1) \uparrow \{\text{failA/B}, \text{advA/B/S}\}. \]

In Figure 3 we illustrate the behaviour of designated receiver B (DR_B in the multicast tree). DR_B has also been specified in terms of three components. A DR behaves like a receiver, except that it may fail and that it advertises itself. DR_B may fail at any time (failB), and enter a state where it stops advertising itself. From this state it may either fatally fail (failB) or recover (recoverB). All actions in DR_B that do not synchronise with its environment are made unobservable, i.e.,

\[ \text{DR}_B = (\text{B-Chnl}B \parallel \text{B-Watch} \parallel \text{Ds-RecB}) \uparrow \{\text{failA/B}, \text{advA/B/S}\}. \]

Note that we have not modelled failure for simple 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 on the receivers are not expected to hold when they fail. Moreover, failures of simple 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 common in the process alphabets. Finally, in our experiments we have not taken into account the behaviour of receivers REC_2 and REC_4, since their behaviour for this part of the protocol is identical to the behaviour of REC_1 and REC_3 respectively.

The following properties have been specified for verification: safety property Right fl refers to REC_1, and states that the receiver is informed about the failure of a DR, if and only if that DR is its selected AP. Liveness property LivRec_3 refers to REC_3, and asserts that when DR_A is the selected AP, REC_3 will select a new AP (selA/S) upon failure of DR_A (ms_failA). In Figure 4 we illustrate these properties as well as the LTS into which they are mapped for analysis purposes.

![Figure 4: Transformation of Properties in Tracta](image)

4. TRACTA AS A TOOL

4.1. The environment

The software architecture of a distributed system can be specified as a configuration of component specifications [11]. Darwin [11] is a declarative configuration language which is intended to be a general
purpose notation for specifying the structure of systems composed from diverse components using
diverse interaction mechanisms. It has both a textual and graphical syntax. Programs are in essence
defined as hierarchic compositions of interconnected components. The following paragraph is an
informal presentation of the features of the Darwin language which directly relate to Tracta.

In Darwin, the component is the basic entity programs are constructed from. Typically, a program
consists of a limited set of component types, with multiple instances of these types. A component type is
defined in terms of its communication interface: services it provides (black circles in Figure 5) to other
components and services it requires (white circles in Figure 5). Component instantiations are declared
within the definition of composite components. Composite components are defined by declaring the
constituent instances of other primitive or composite components and the bindings between those
components. The binding of a service requirement to a service provision is type-safe, according to the
communication mechanisms employed at the two sides and the communicated data. The compatibility
test invoked is determined by the target distributed systems platform. Requirements which cannot be
satisfied inside the composite component are made visible at a higher level. Similarly, services provided
internally are bound to an interface service provision, when they are required from outside.

![Figure 5: REC_1 in Darwin, using the SAA](image)

The Software Architect's Assistant (SAA) [23] is a visual environment for the design and development
of distributed programs. It provides users with automated assistance throughout the software design
process. Facilities provided include the display of multiple integrated graphical and textual views, a
flexible mechanism for recording design information and the automatic generation of program code and
formatted reports from design diagrams. The SAA supports Darwin as a configuration language. The
Regis execution platform [24] is currently used for a system developed in this way.

Figure 5 illustrates the Darwin configuration of component REC_1, described in section 3. REC_1 is a
composite component, made up of the Receiv_1, Chnl_1, and Watch_1 components. The bindings
among these reflect synchronised communication, and are labelled in the same way as their
respective LTSs. Bindings with components at other levels can also be represented, as in the case of action advA/B/S of the Chnl_1 component. Due to the fact that our
components have been especially designed for our case study, the names of the service provisions and
requirements bound together are compatible. In the general case, however, components may be reused
from component libraries. In that case, a binding in Darwin will correspond to renaming the
corresponding labels in transitions of the LTSs for the components to a common name, for synchronisation purposes. Moreover, services that are not visible to other components such as mesA/B/S,
component Receiv_1, correspond to actions made internal in Tracta.
A one-to-one mapping between the features of Darwin and the concepts defined in Tracta is not possible in all cases. The Darwin language can be used to describe generic system architectures from which instances can be generated. Analysis will be performed after instantiation in such cases. Moreover, a binding in Darwin does not need to correspond to a single action in LTS behaviour. Intervention is then required by the user for correct renaming of actions to be synchronised.

4.2. Implementation

The Tracta tool for analysing the behaviour of distributed systems has been implemented in C++ and is running on Unix. The main functionality of analysis has been implemented within classes LTS and Composite. The latter defines objects that correspond to composite subsystems and contain the information required for performing composition of LTSs. Every incremental analysis step is made in four stages that correspond to the modules illustrated in Figure 6.

![Figure 6: Tracta in its Environment](image)

**Module 1 - Construct:** Constructs an instance $C$ of class Composite for a subsystem in the compositional hierarchy. The constructor of the class reads from a file the specifications for the composite component, i.e. what are its immediate children in the compositional hierarchy, which ones of those, if any, are safety or liveness properties, what actions are to be hidden from the level above. An array of pointers to LTS objects is created for $C$. These objects have been constructed from specifications in their respective files. Properties are a subclass of the LTS class. Their constructor additionally transforms them as required for analysis purposes, transparently to the user. $C$ thus contains all the necessary information for composition to be performed.
pot_act : list of Potential_act;
cart_prod : list of composite states;
W = (initial state of composite component);
while (W non-empty) do {
    q = head of W;
pot_act = find_potent_act(q);
    ∀ a in pot_act do {
        cart_prod = cartesian_product(a);
        ∀ s in cart_prod do {
            if (s has id) { /* means s not in (W ∪ A) */
                give id to s;
                add s to W;
            } /* if */
            add transition (id(q), a, id(s)) to resulting LTS;
        } /* ∀ s */
    } /* ∀ a */
} /* while */

Figure 7: The Composition Algorithm

Module 2 - Compose: This module is implemented by a method of class Composite. For an object C belonging to class Composite, the method returns the LTS object that corresponds to the composition of the immediate children of C in the hierarchy. The algorithm for composition is illustrated in Figure 7. As described in [13], composition uses two sets of states, a working set W of system states to be explored, and a set A of states already explored. A hash table is also used for storing the ids assigned to states of the composite system that have already been visited. In our implementation, a database of actions appearing in the components of a composite object is generated at the object construction stage. The number of component LTS that need to synchronise on each action is additionally stored. In the algorithm of Figure 7, Potential_act is a class with the following data members: an action id, an integer denoting how many LTS remain to synchronise on this action, and an array containing for each LTS L a pointer to_states where this action can lead, starting from the state of L in q. Procedure find_potent_act returns the list of Potential_acts for which how_many has become zero, i.e. of those actions that can be performed from q. For each potential action a in pot_act, the cartesian product of the sets of states in a to_states corresponds to the composite LTS states that are reachable from q with a.id.

Module 3 - Minimise: Module minimise is optional. It is invoked when the user requests minimisation to be performed after composition. It can also be used on a given LTS irrespective of composition. For a given LTS L, the minimised version of L with respect to weak/strong equivalence is returned, depending on the user’s requirements. Due to the fact that minimisation takes up most of the computational effort in our analysis method, we have implemented the algorithm presented by Fernandez [25]. In general, strong equivalence [20], can be tested in O(m n) time for a labelled transition system with m transitions and n states. However, the problem can be reduced to the relational coarsest partition problem, which has been solved by Paige and Tarjan [26] 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 = <S, A, Δ, q0> with respect to weak equivalence, is reducible to minimising P’ = <S, A, Δ’, q0> with respect to strong equivalence, where Δ’ is obtained from Δ by making its τ-relation closed under reflexivity and transitivity [27]. For those interested, the implementation of the minimisation algorithm as well as of the design of the Tracta tool will be described in detail in a report under preparation.

Module 4 - Analyse: This module returns information obtained from stages 2-3. More specifically it reports timing results for the algorithms of composition and minimisation, the sizes of graphs for the subsystem after composition and after minimisation, as well as deadlocks that may have been detected for the composite LTS (see Figure 9). We are currently working on providing additional information at
the final stage of analysis, i.e. for the root of the compositional hierarchy. We intend to notify the user of property violations. Identifying the property violated is simple in Tracta, but it is equally important to return a counterexample to demonstrate how the violation may occur. Work in this area is currently in progress [4].

The way Tracta currently interacts with its environment is illustrated in Figure 6. The SAA is the visual environment that intends to integrate tools for monitoring, analysis and reverse engineering through a common interface. The SAA is used for the design of a system architecture which is used by the Darwin compiler to generate a system instance. The hierarchical structure of a system instance can be utilised for analysis. A graphical user-interface will be provided for Tracta, which we plan to include in the new version of the SAA currently being implemented in Java. At present an initial user-interface, SEAL [28], has been built in Tcl/Tk specifically for the purpose of analysis.

Figure 8 illustrates the use of SEAL for our case study described in section 3. SEAL consists of two windows, the graphical LTS editor, and the window with the system decomposition tree. Selecting a primitive component in the hierarchy results in the LTS editor being ready to receive the LTS specification for it. In the figure, component Chnl_1 has been selected and its LTS specification has been provided in the SEAL LTS editor. The user proceeds similarly for all primitive components. The results returned from the tool are then displayed in the “Analysis Report” window, as shown in Figure 9. Figure 9 displays the analysis results for component AP_Select in the RMTP hierarchy of Figure 8.

![Figure 8: The SEAL User-Interface](image-url)
The checking of property violation is currently performed by referring to the global graph for existence of the $\pi$ state, or of terminal sets of states where acceptance transitions of liveness properties do not appear. For example, the graph of Figure 9 contains no $\pi$ state which proves that safety property Right_fl is not violated by the system. Furthermore, terminal sets of states contain acceptance transitions for property Liv_Rec3, which proves that the property is satisfied.

We have used the same case study in [3], where our approach helped us to uncover a mistake in our original specifications, when we found that one of the properties that we had introduced was being violated. We have also found that Tracta has concluded in 4 states and 9 transitions for the final graph (Figure 9) as compared to 96,528 states and 672,588 transitions which conventional reachability analysis results in. In [3] we have shown that Tracta may achieve a considerable reduction to the size of LTS that it needs to handle, even as compared to conventional Compositional Reachability Analysis, due to the fact that it does not need to expose at the higher levels actions that concern properties which need to be checked.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have described the Tracta approach for analysing the behaviour of distributed systems. We have briefly introduced the modelling and verification aspects of the approach. A case study of a reliable multicast transport protocol has been used for illustrating the methodology. The focus of this paper, however, has been on presenting Tracta as an automated tool which forms an integral part of an environment for the design and implementation of distributed systems. We have thus demonstrated that analysis should not be an isolated part of the construction of distributed software. Rather, it can play a significant role in the design process since it enforces clear modelling besides providing verification capabilities.

The great asset of Tracta is that it has been specifically designed to suit the hierarchical composition of components which is used in the development of distributed systems. Moreover it provides to the designer a uniform way of modelling both properties and processes in terms of state-machines, and of handling them by just including them in the compositional hierarchy. Transformations to properties are performed transparently to the designer. Finally, besides returning useful results about the system under analysis, Tracta generates an abstracted view of the system state-graph according to the user specifications.

We are currently working on improving the results that the Tracta tool returns. Our immediate implementation plans include: firstly, a method for keeping track of safety properties being violated [4], secondly, the algorithm proposed by Aho et al. [29] for detecting strongly connected components for the case of liveness properties. We are also investigating the approach proposed by Yeh [30] for
reconstructing counterexample traces in the presence of action hiding. Our aim is to provide system designers with significant information and guidelines concerning their designs. In the long term, Tracta is to be integrated in our distributed systems development environment through the Software Architect's Assistant.

REFERENCES


[14] W. J. Yeh, "Controlling State Explosion in Reachability Analysis," : SERC, Purdue University, 93.


