Chapter

Behaviour Analysis of Software Architectures

Jeff Magee, Jeff Kramer and Dimitra Giannakopoulou
Department of Computing, Imperial College of Science, Technology and Medicine

Key words: Software Architecture, behaviour Analysis

Abstract: The overall structure of a system described by a set of components and their interconnections is termed its software architecture. In this paper, we associate behavioural specifications with components and use these specifications to analyze the overall system architecture . The approach is based on the use of Labelled Transition Systems to specify behaviour and Compositional Reachability Analysis to check composite system models. The architecture description of a system is used directly in the construction of the model used for analysis. Analysis allows a designer to check whether an architecture satisfies the properties required of it. The paper uses examples to illustrate the approach and discusses some open questions arising from the work.

1. INTRODUCTION

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. The initial emphasis in Software Architecture specification has thus been in capturing system structure [5,8,13]. The authors have previously published papers on the use of the architecture description language Darwin for specifying the structure of distributed systems and subsequently directing the construction of those systems [8,9,10]. Darwin can also be used to organise CORBA based distributed systems [11]. Darwin describes a system in terms of

components, which manage the implementation of services. Interconnection structure is specified by bindings between the services required and provided by component instances. Darwin has both a graphical and a textual form with appropriate tool support [9,12].

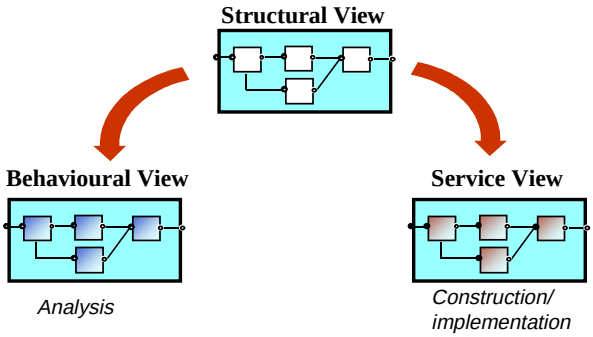


Figure 1. Common Structural View with Service and Behavioural Views

In this paper, we describe the use of Darwin structural descriptions as a framework for behaviour analysis rather than system construction. Darwin has been designed to be sufficiently abstract to support multiple views (cf. [7]), two of which are the behavioural view (for behaviour analysis) and the service view (for construction) (Figure 1). Each view is an elaboration of the basic structural view: the skeleton upon which we hang the flesh of behaviour specification or service implementation [14].

In previous papers, we have discussed the use of Darwin to produce the service view, with components providing and requiring services at their interfaces and with implementation definitions for the primitive components. For example, when used to structure CORBA systems [11], the computational behaviour of Darwin primitive components is determined by CORBA object implementations and these object implementations interact via interfaces specified in IDL using the ORB in the usual way. Primitive components encapsulate objects and specify their instantiation, their required interfaces and provided interfaces. As depicted in figure 2, a primitive component may embed one or more objects.

In this paper we concentrate on the behavioural view using Labelled Transition Systems (LTS) for behaviour specification and analysis. The analysis approach is Compositional Reachability Analysis CRA [4]. We have developed techniques for analysing system models in the CRA setting with respect to both safety [2] and liveness [3] properties. The techniques are supported by software tools, which provide for automatic composition, analysis, minimisation, animation and graphical display. We first describe the relationship between components and their behavioural specifications.

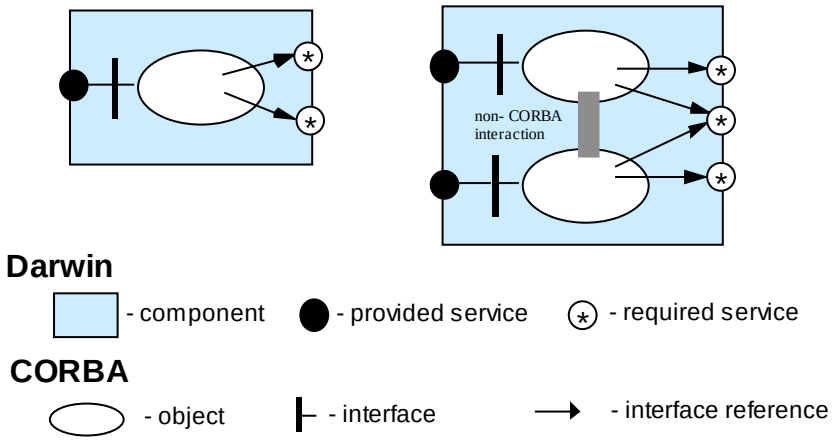


Figure 2. Embedding objects in components

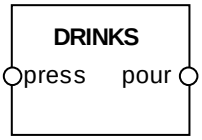
2. PRIMITIVE COMPONENTS

A primitive component is one with no substructure of components. In the service view of architecture, a primitive component has an implementation defined by an object or objects programmed in a programming language such as C++. In the behavioural view, a primitive component is defined as a finite state LTS. The example of figure 3 depicts the Darwin graphical and textual description of a primitive component with two interfaces.

In the behavioural view, we do not distinguish between provided and required services, service access points are simply declared as *portals*. Consequently, implementation details such as invocation direction can be deferred, although, in many cases, it is obvious from the behavioural model as to which component is providing a service and which is using it.

A major objective of our work in architectural analysis is to provide tools that are both accessible and usable by practising engineers. To this end, we originally conceived that the behaviour of primitive components should be specified graphically as state transition diagrams since these should be familiar in one form or another to software engineers. However, it quickly became apparent that this is an extremely cumbersome method for other than trivial behaviour specifications. With our focus on actions rather than states in specifying behaviour, it was natural to use process algebra as a concise notation for describing behaviour. However, it is unlikely that most software engineers have a working knowledge of process algebra. To mitigate this

problem, we have included the facility to depict textual specifications as labelled transition diagrams. These diagrams may be animated, by an interactive behaviour simulation, to check that the specification corresponds to the engineer's intuition.



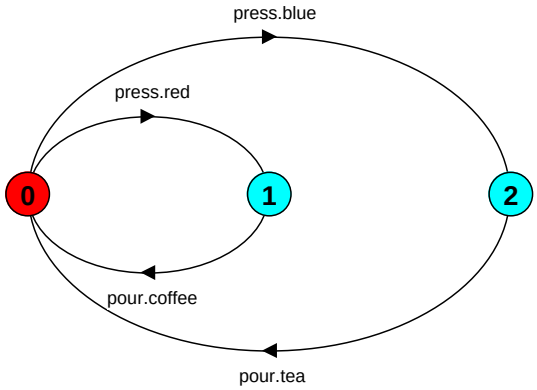

```
interface BUTTON  {red; blue;}

interface BEVERAGE{coffee; tea;}

component DRINKS {
  portal press:BUTTON;
  portal pour :BEVERAGE;
}
```

Figure 3. Darwin description of DRINKS component

The behaviour of the drinks component is modelled in Figure 4 both graphically as a Labelled Transition System and textually in our process algebra notation FSP (Finite State Processes).




```
DRINKS = (press.red  -> pour.coffee -> DRINKS
         |press.blue -> pour.tea    -> DRINKS
         ) @ { press, pour}.
```

Figure 4. Behavioural description of DRINKS component

Primitive components are defined as finite state processes in FSP using action prefix "`->`" and choice "|". If `x` is an action and `P` a process then `(x->P)` describes a process that initially engages in the action `x` and then behaves exactly as described by `P`. If `x` and `y` are actions then

`(x->P|y->Q)` describes a process which initially engages in either of the actions `x` or `y`. After the first action has occurred, the subsequent behaviour is described by `P` if the first action was `x` and `Q` if the first action was `y`. Thus the `DRINKS` component offers a choice of the actions `press.red` and `press.blue`. As a result of engaging in one of these actions the appropriate drink is poured. The behavioural view does not distinguish between input and output actions although, as in the example, input actions generally form part of a choice offered by a component while output actions do not. The `@{press,pour}` states that all actions labelled or prefixed by `press` or `pour` can be shared with other components. The next example is a component that has internal actions that cannot be shared with other components. Figure 5 gives the Darwin graphical description for the primitive component `LOSSYCHAN` together with its behaviour modelled in FSP and the corresponding LTS diagram.

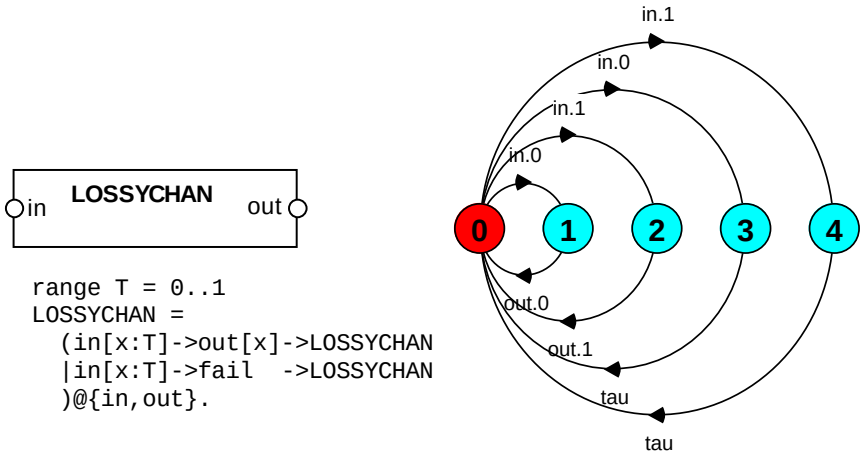
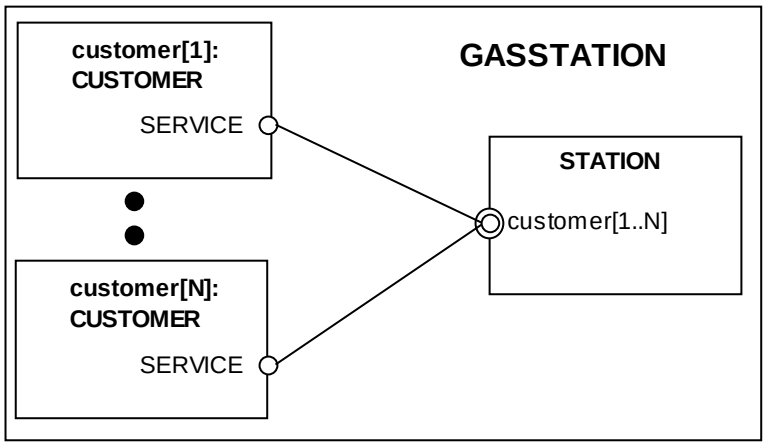


Figure 5. LOSSYCHAN component

The component `LOSSYCHAN` models a channel which inputs values in the range `0..1` and then either outputs the value or fails. In other words, the component models a transmission channel that can lose messages. Failure is modelled by non-deterministic choice on the input, which leads to the internal action `fail`, if failure is chosen. Since `fail` does not appear at the interface of the component, it becomes the silent action `tau` in the LTS diagram for the component. In many Architectural Description Languages, `LOSSYCHAN` would be represented as a connector rather than a component [1,13]. However, Darwin does not have a separate connector construct. Connectors can be distinguished as a particular class of components. It is clear from the above that connectors are modelled in exactly the same way as components.

The modelling notation FSP – Finite State Processes – includes guarded choice, local processes and conditional processes. However, these are syntactic conveniences to allow concise model definition. Definitions using these constructs can all be expressed using action prefix, choice and recursion as described in this section.




```
const int N = 3; //#customers

interface SERVICE {
  prepay(int); gas(int);
}
component CUSTOMER {
  portal
    SERVICE;
}
component STATION {
  portal
    customer[1..N]:SERVICE;
}
```

```
component GASSTATION {
  inst
    STATION;
  forall i = 1 to N {
    inst
      customer[i]:CUSTOMER;
    bind
      customer[i].SERVICE
       --STATION.customer[i];
  }
}
```

Figure 6. GASSTATION composite component

3. COMPOSITE COMPONENTS

A composite component is constructed from interconnecting instances of more primitive components. A composite component defines a structure and no additional behaviour. Its behaviour can therefore be computed based on this structure and the behaviour of its components. To illustrate composition, we will use the Gas Station problem, originally stated in [16] and more

recently addressed in [2,17]. The Gas Station problem concerns a set of *N* customers who obtain gas by prepaying a cashier who activates one of *M* pumps to serve the customer. The overall `GASSTATION` component is depicted in figure 6.

In an implementation such as CORBA discussed in the introduction, Darwin bindings (drawn as arcs between portals) are generally references to objects. In the behavioural view, a binding denotes an action shared between two components. Each customer in figure 6 shares the actions `prepay` and `gas`, which constitute the `SERVICE` interface, with the `STATION` component. Component instances in the behavioural view are finite state processes as described in the previous section. The composite behaviour is the parallel composition of these processes. Consequently, the behaviour of `GASSTATION` is the parallel composition of its constituent components:

```
||GASSTATION = (customer[1..N]:CUSTOMER || STATION).
```

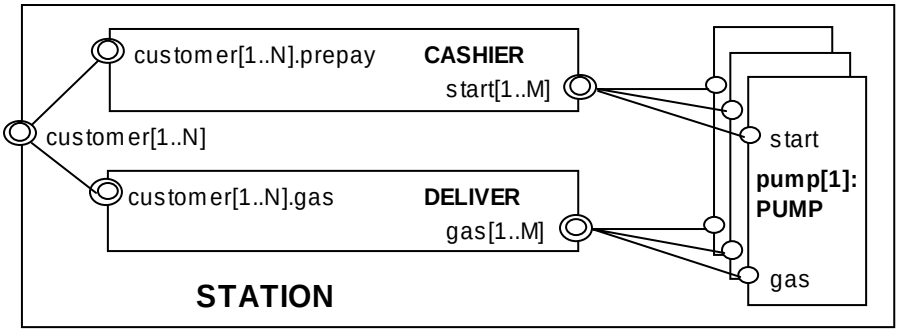
Note that to create multiple copies of `CUSTOMER` we use process labelling. Each action label of the customer process (namely `prepay` and `gas`) is prefixed with the process label. Thus customer 1 has the action labels `customer[1].prepay` and `customer[1].gas`. The `STATION` is itself a composite component consisting of the cashier and one or more pumps as depicted in figure 7. A `DELIVER` component is also required to associate pump actions with customer actions. The need for this component is discussed later in the paper.

A binding in Darwin always denotes a shared action in the behavioural view. Shared actions are the means by which processes synchronise and interact in FSP. It is sometimes necessary to relabel actions to ensure that the shared action has the same name in all the processes that share that action. Relabelling is required in the FSP description of the `STATION` component based on the particular bindings:

```
||STATION = (CASHIER || pump[1..M]:PUMP || DELIVER)
            /{pump[i:1..M].start/start[i],
              pump[i:1..M].gas/gas[i]}
             @{customer}.
```

The general form of the relabeling function is:

/{*newlabel*_1/*oldlabel*_1,… *newlabel*_n/*oldlabel*_n}.




```
const M = 2; //#pumps
component STATION {
  portal customer[1..N]:SERVICE;
  inst CASHIER;
  inst DELIVER;
  forall i = 1 to N bind
     customer[i].prepay -- CASHIER.customer[i].prepay;
     customer[i].gas    -- DELIVER.customer[i].gas;
  forall i = 1 to M {
    inst pump[i]:PUMP;
     bind
       pump[i].start -- CASHIER.start[i];
       pump[i].gas   -- DELIVER.gas[i];
  }
}
```

Figure 7. `STATION` composite component

This section has outlined how the FSP composition expressions for the behavioural model can be generated directly from the Darwin composite component structure. In the next section, we discuss analysis using the behavioural model.

4. ANALYSIS

The complete behavioural model for the Gas Station is listed in figure 8. It includes behaviour definitions for the primitive components, `CUSTOMER`, `CASHIER`, `PUMP` and `DELIVER`. A `CUSTOMER` makes a prepayment of some amount (`a`) chosen from the range (`A`) and then inputs some amount of gas (`x`). The process definition includes a test to check that the amount of gas actually delivered is the same as the amount paid for. In this simplified

model of the Gas Station, the cashier does not give change and pumps are expected to deliver the amount of gas that has been paid for. The CASHIER starts any pump that is ready and passes to it the identity of the customer (`c`) and the amount of gas required (`x`). The `PUMP` outputs the correct amount of gas, which is delivered to the `CUSTOMER` by the `DELIVER` component. The composition expressions for the composite components `STATION` and `GASSTATION` are as described in the previous section.

```
const N = 3     //number of customers
const M = 2     //number of pumps
range C = 1..N  //customer range
range P = 1..M  //pump range
range A = 1..2  //amount of money or Gas

CUSTOMER = (prepay[a:A]->gas[x:A]->
                if (x==a) then CUSTOMER else ERROR).
CASHIER =
  (customer[c:C].prepay[x:A]->start[P][c][x]->CASHIER).
PUMP =
  (start[c:C][x:A] -> gas[c][x] -> PUMP).
DELIVER=
  (gas[P][c:C][x:A] -> customer[C].gas[x] -> DELIVER).

||STATION = (CASHIER || pump[1..M]:PUMP || DELIVER)
            /{pump[i:1..M].start/start[i],
              pump[i:1..M].gas/gas[i]} @{customer}.

||GASSTATION =  (customer[1..N]:CUSTOMER ||STATION).
```

Figure 8. Gas Station Behavioural Model

Animation

Our analysis tool *LTSA* (Labelled Transition System Analyser) allows a user to explore different execution scenarios using the behavioural model. To do this, the user must specify the set of actions that he/she wants to control. The controlled set of actions is defined by a menu, which for figure 9 is:

```
menu RUN = {customer[C].prepay[A]}
```

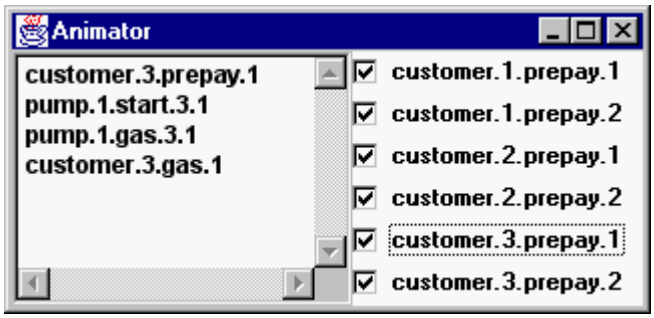


Figure 9. Animating the Gas Station

Figure 9 depicts the trace of actions which result from instigating a prepay action from customer 3. The cashier allocates pump 1, which delivers the requisite gas to the customer via the `DELIVER` process.

Reachability Analysis

Animation allows a user to explore different execution scenarios, however, it does not allow general properties concerning the model to be checked. For example, does a customer *always* receive the correct amount of gas? Reachability analysis performs an exhaustive search of the state space to detect `ERROR` and deadlock states (no outgoing transitions). In fact the behaviour model of figure 7 has a bug which permits incorrect behaviour. The output of the analyser is shown below:

```
property customer.3:CUSTOMER violation.
property customer.2:CUSTOMER violation.
property customer.1:CUSTOMER violation....
States Composed: 3409 Transitions: 11862 in 1468ms
Trace to property violation in customer.2:CUSTOMER:
customer.1.prepay.1
pump.1.start.1.1
customer.2.prepay.2
pump.1.gas.1.1
customer.2.gas.1
```

The output shows that a property violation in each of the customer components is detected. In addition, an example trace, which causes one of the violations, is produced. Remembering that the `CUSTOMER` model requires that the amount of gas delivered to the customer should be the amount paid for, the trace is an execution in which customer 2 gets the gas paid for by customer 1. This error is essentially the same as the race condition discussed in [17]. The error in the model is that the `DELIVER`

process delivers gas to *any* ready customer C rather than to the customer identity c passed to it by the cashier. The corrected DELIVER process is:

```
DELIVER
   =(gas[P][c:C][x:A] -> customer[c].gas[x] -> DELIVER).
```

Safety properties

We can specify safety properties that a composition of components must satisfy using property automata [2]. These specify the set of all traces that satisfy the property for a particular action alphabet. If the model can produce traces, which are not accepted by the property automata, then a violation is detected during reachability analysis. For example, the following automaton specifies that for, two customers, if one customer makes a payment then he or she should get gas before the next customer makes a payment. In other words, service should be FIFO.

```
range T = 1..2
property
   FIFO      = (customer[i:T].prepay[A] -> PAID[i]),
   PAID[i:T] = (customer[i].gas[A]      -> FIFO
               |customer[j:T].prepay[A] -> PAID[i][j]
               ),
   PAID[i:T][j:T] = (customer[i].gas[A] -> PAID[j]).
```

A Gas Station with a single pump satisfies this property, however, a station with two pumps does not and leads to the following violation:

```
Composing
 property FIFO violation.
States Composed: 617 Transitions: 1398 in 94ms
Trace to property violation in FIFO:
customer.1.prepay.1
pump.1.start.1.1
customer.2.prepay.1
pump.2.start.2.1
pump.2.gas.2.1
customer.2.gas.1
```

The trace describes the scenario in which customer 1 pays first and gets pump 1 followed by customer 2 paying and getting pump 2. Clearly in a two pump system, pump 2 can finish first, thereby violating the FIFO property.

Liveness properties

The LTSA analysis tool allows behavioural models to be checked against specific liveness properties specified in Linear Temporal Logic. However,

we have found a check for a general liveness property which we term *progress* to provide sufficient information on liveness in many examples. Progress asserts that in an infinite execution of the system being modelled, all actions can occur infinitely often. In the gas station example, it would assert that customers will always eventually be served. In performing the progress check, we assume fair choice which means that if an action is eligible infinitely often, then it is executed infinitely often. With this assumption, the progress check finds no problem with the gas station. However, we can examine the behaviour of the system under different scheduling constraints by applying action priority. For example, the system below states that the actions of customer 1 have lower priority than other actions:

```
||GASSTATION = (customer[1..N]:CUSTOMER ||STATION)
                >>{customer[1]}.
```

Unsurprisingly, this causes a progress check violation since it is now possible for the cashier to ignore customer 1 in favour of other customers. Customer 1 may never be served. The tool gives the following output.

```
Progress violation for actions:
  {customer.1.prepay.1, customer.1.gas.1, customer.1.gas.2,
customer.1.prepay.2, pump.1.start.1.1, pump.2.start.1.1,
pump.1.start.1.2, pump.2.start.1.2, pump.1.gas.1.1,
pump.1.gas.1.2..........}
   Trace to terminal set of states:
   Actions in terminal set:
   {customer.2.prepay.1, customer.2.gas.1, customer.2.gas.2,
customer.2.prepay.2, customer.3.prepay.1, customer.3.gas.1,
customer.3.gas.2, customer.3.prepay.2, pump.1.start.2.1,
pump.2.start.2.1..........}
```

This includes the set of actions that do not occur infinitely often in the system and the set of action that can occur infinitely often. It is clear that actions for customer 1 occur in the former set and the actions for customer 2 in the latter. The tool gives a trace that leads to the execution in which the violation occurs. In the example, this trace is empty, as customer 1 never gets an opportunity to get gas.

5. DISCUSSION & CONCLUSIONS

We have presented an approach that associates behaviour descriptions with architectural components and supports behaviour analysis of the composition of these components according to the software architecture. Although relatively small, the example exhibits non-trivial behaviour. It demonstrates that we can produce concise and flexible behavioural models in which it is easy to add additional components and interactions. In the Gas Station, it is trivial to modify the numbers of customers and pumps. In fact, the Gas Station as presented is an instantiation of a common distributed software architecture style known as a multi-server or multithreaded server. In a multi-server system, a separate server thread allocated by an administrator thread handles each client request.

In the introduction we stated that we could use the same structural description for system construction as for behaviour modelling. This is not always the case. For example, the Gas Station behavioural view includes the `DELIVER` component which routes pump actions to customers. This component would not appear in the service view since this routing would be implicit in the service invocation mechanism. `DELIVER` is modelling an aspect of architectural connection and it is specific to the behavioural view. In other words, we recognise that there is a need to augment the structural description with connector components for the behavioural view of architecture. In contrast to Wright [1] we have resisted requiring that a connector component is *always* interposed between application components since this seems to lead to large numbers of auxiliary actions.

An issue that always arises when considering exhaustive state space search methods is scalability. We have used the current toolset, which has not yet been optimised for performance, to analyse an Active Badge System[21] in which the final model has 566,820 reachable states and 2,428,488 possible transitions. This took 400 seconds to construct and check on a 200MHz Pentium Pro and required 170Mb of store. Although not addressed in this paper, our tools support compositional reachability analysis in which intermediate composite components can be minimised with respect to their interface actions using observational equivalence. Previous work [15] has addressed the problem of intermediate state explosion.

We believe that analysis and design are closely inter-linked activities which should proceed hand in hand. The FSP notation and its associated analysis tool *LTSA* have been carefully engineered to facilitate an incremental and interactive approach to the development of component

based systems. Analysis and animation can be carried out at any level of the architecture. Consequently, component models can be designed and debugged before composing them into larger systems. The analysis results are easily related to the architectural model of interconnected components. The *LTSA* analysis tool described in this paper is written in Java™ and can be run as an application or applet. It is available at http://www-dse.doc.ic.ac.uk/~jnm. The approach we have described in this paper to analysing component-based systems is a general one that is not restricted to a particular tool-set. For example, CSP/FDR [6,19] has been used with the architectural description language Wright[1] and both LOTOS/CADP [18] and Promela/SPIN [20] have been used in the context of analysing software architectures. The objective, whatever the tool, is to use behaviour analysis during design to discover architectural problems early in the development cycle.

References

[1] Allen R. and Garlan D., *A Formal Basis for Architectural Connection*, ACM Transactions on Software Engineering Methodology TOSEM, 6 (3), July 1997, 213-249.

[2] Cheung S.C. and Kramer J., *Checking Subsystem Safety Properties in Compositional Reachability Analysis*, (18th IEEE Int. Conf. on Software Engineering (ICSE-18), Berlin, 1996), 144-154.

[3] Cheung S.C., Giannakopoulou D., and Kramer J., *Verification of Liveness Properties using Compositional Reachability Analysis*, (6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 97), Zurich, Sept. 1997), LNCS 1301, (Springer-Verlag), 1997, 227-243.

[4] Giannakopoulou D., Kramer J. and Cheung S.C., *"Analysing the Behaviour of Distributed Systems using Tracta,"* Journal of Automated Software Engineering, special issue on Automated Analysis of Software (to appear), vol. 6(1). R. Cleaveland and D. Jackson, Eds.

[5] Garlan D. and Perry D.E., *Introduction to the Special Issue on Software Architecture*, IEEE Transactions on Software Engineering, 21 (4), April 1995, 269-274.

[6] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, N.J., 1985.

[7] Kruchten P.B., *The 4+1 Model of Architecture*, IEEE Software, 12 (6), Nov. 1995, 42-50.

[8] Magee J., Dulay N., Eisenbach S., Kramer J., *Specifying Distributed Software Architectures*, (5th European Software Engineering Conference (ESEC '95), Sitges, September 1995), LNCS 989, (Springer-Verlag), 1995, 137-153.

[9] Magee J., Dulay N. and Kramer J., *Regis: A Constructive Development Environment for Distributed Programs*, Distributed Systems Engineering Journal, 1 (5), Special Issue on Configurable Distributed Systems, (1994), 304-312.

[10] Magee J. and Kramer J., *Dynamic Structure in Software Architectures*, (4th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 4), San Francisco, October 1996), SEN, Vol.21, No.6, November 1996, 3-14.

[11] Magee J., Tseng A., Kramer J., *Composing Distributed Objects in CORBA*, (Third International Symposium on Autonomous Decentralized Systems (ISADS 97), Berlin, Germany, April 9 - 11, 1997.

[12] Ng K., Kramer J. and Magee J., *Automated Support for the Design of Distributed Software Architectures*, Journal of Automated Software Engineering (JASE), 3 (3/4), Special Issue on CASE-95, (1996), 261-284.

[13] Shaw M., et al., *Abstractions for Software Architecture and Tools to Support Them*, IEEE Transactions on Software Engineering, 21 (4), April 1995, pp 314-335.

[14] Kramer J. and Magee J., *Exposing the Skeleton in the Coordination Closet*, (2nd IEEE International Conference on Coordination Models and Languages, Coord '97, Berlin , September 1997), LNCS 1282, (Springer-Verlag), Sept 1997, pp. 18-31.

[15] Cheung S.C. and Kramer J., *Context Constraints for Compositional Reachability Analysis*, ACM Transactions on Software Engineering Methodology TOSEM, 5 (4), (1996), 334-377.

[16] Hembold, D. and Luckham, D.C., *Debugging Ada Tasking Programs*, IEEE Software, 2(2), March 1985, 47-57.

[17] Naumovich, G., Avrunin G.S., Clarke L.A. and Osterweil L.J. *Applying Static Analysis to Software Architectures*, (6th European Software Engineering Conference / 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 97), Zurich, Sept. 1997), LNCS 1301, (Springer-Verlag), 1997, 77-93.

[18] Jean-Pierre Krimm and Laurent Mounier. Compositional state space generation from LOTOS programs. In Ed Brinksma, editor, Proceedings of TACAS'97 (Tools and Algorithms for the Construction and Analysis of Systems), Enschede, The Netherlands, April 1997. Springer Verlag.

[19] Formal Systems, *Failues Divergence Refinement: FDR 2.0 User Manual*, ed. Formal Systems (Europe), Oxford, U.K, August 1996.

[20] Holtzman G.J., *The Model Checker SPIN*, IEEE Transactions on Software Engineering, 23(5) May 1997, 279-295.

[21] Magee J., Kramer J. and Giannakopoulou D., *Analysing the Behaviour of Distributed Software Architectures: a Case Study*, 5th IEEE Workshop on Future Trends in Distributed Computing Systems, FTDCS'97, Tunsia, October 1997.