Modelling for mere Mortals

Jeff Kramer and Jeff Magee

Department of Computing
Imperial College of Science, Technology and Medicine,
London SW7 2BZ, UK.
{jk,jnm}@doc.ic.ac.uk

Abstract. In the past, attempts to convince practising software engineers to adopt formal methods of software development were generally unsuccessful. The methods were too difficult to learn and use, provided inadequate tool support and did not integrate well into the software development process. In short, they could only be used effectively by the gods who created them! Are we in a better position today? Recent advances in and experience with specification techniques and automated model checking have demonstrated the utility of these techniques. In this paper we outline one such effort which is specifically intended to facilitate modelling as part of the software process, and to try to make model specification and model checking accessible to mere mortals.

1 Introduction

The ACM 50th Anniversary edition of Computing Surveys contains two excellent papers describing the state of the art and research directions in Formal Methods [1] and Concurrency [2]. The former paper describes the failure of past formal methods to make a real impact on practising software engineers, indicating that they were just too difficult to learn and use, provided inadequate tool support and did not integrate well into the software development process. The paper describes how recent advances in fundamental research and improved technology has enabled a number of specification and verification techniques to be used in practice on major industrial case studies. However, if we are to make a significant impact on practice, we must still seek to make our methods and tools more attractive and accessible to practioners. In addition, we need to provide the associated education and technology transfer.

The latter paper [2] provides an excellent overview of the issues and current research in the field of concurrency. Again, the need for education, technology transfer and improved integration with the software lifecycle is identified. Furthermore, the paper states: "Traditionally, software engineering devotes much attention to organizational and procedural issues in software development and relatively little to methods for system analysis; in this respect it resembles a management discipline rather than an engineering one. Tools based on concurrency theory offer a particularly appropriate starting point for putting the engineering into software engineering."

One of the main aims of our research is to provide sound and accessible techniques for modelling and model verification associated with the development of concurrent and distributed systems. We are acutely aware of the problems of advocating powerful but erudite approaches usable by only the developers themselves. We have therefore sought to adopt and adapt concepts and techniques which offer the best hope of widescale use by ordinary, competent engineers. We recognise that no one method or tool will suffice for all purposes, and believe that it is better to provide a well-focussed usable approach than one which is more general but less usable. Hence we focus our work on the modelling of only a particular aspect of a system – concurrent behaviour.

Integration with the software development process

Our approach exploits the software architecture as the underlying design structure of a system, common to the various phases of system development. In particular, we use the Darwin architecture description language [3,4] which has been designed to be sufficiently abstract to support multiple views (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 [5]. The service view describes the system as a hierarchical composition of components, each of which provides and requires services at its interface, with implementation elaborations for the primitive components. The behavioural view models the system as a hierarchical, parallel composition of component processes, each of which interacts with other processes via shared actions at its interface, with behaviour elaborations for the primitive component processes. In essence, the architecture drives the process of putting together individual component specifications or implementations in order to obtain a system with desirable characteristics. When performing analysis, these characteristics are formally described in terms of properties against which the specified system is checked.

Fig. 1. Common Structural View, with Service and Behavioural Views

Specification and Analysis

State machines are a popular modelling technique which is widely taught and used. For this reason, we were attracted to the use of Labelled Transition systems (LTS) as the underlying formalism for our work. For the verification of finite models, model checking offers a fast, automatic technique with the benefit of providing counterexamples as feedback when property violations are detected. This satisfies our belief that automated tools are essential to aid verification. Furthermore, LTS supports the appropriate compositionality (using Compositional Reachability Analysis CRA) with the components specified simply as finite state processes (FSP) [6]. Compositional methods are desirable in the way that they can reflect the structure of the system. In addition, we have techniques for ameliorating the problem of exponential state explosion in some circumstances [7] and for analysing for both safety [8] and liveness [9] properties. Our property checking mechanisms have been specifically designed for our models that focus on actions rather than states and also address issues related to CRA techniques. As liveness property checks can be expensive, we have also identified a subclass of such properties that occur frequently in practice, and which can be checked directly on the graph of the system, without the use of Büchi automata [10]. This class has been named *progress*. Finally, our methods also support action priority, which allows users to concentrate on specific parts of system behaviour, to impose adverse conditions, or perform a partial search when an exhaustive search cannot be achieved.

All this is supported by the LTS Analyser (LTSA) which provides for automatic composition, analysis, minimisation, animation and graphical display [5,11,12].

Paper outline

Rather than describe the underlying theory or details of the analysis techniques, we use an example to illustrate our general approach and tool support. In particular, we use a Supervisor-Worker/Tuple Space example of a concurrent architecture to illustrate the use of FSP/LTS and the LTSA tools for specification and reasoning. For the sake of brevity, we make very little attempt to compare our work with that of others. The Formal Methods and Concurrency papers [1,2] provide an excellent survey of the field and of related work. Instead we merely try to indicate the reasons for some of our choices. We introduce only as much of the notation and analysis techniques as necessary for the example. Full details can be found in [13] and at the web site: <http://www-dse.doc.ic.ac.uk/concurrency/>.

2. Architectural Design and Analysis: an Example

2.1 Supervisor-Worker Description

Supervisor-Worker is a concurrent architecture that can be used to speed up the execution of some computational problems by exploiting parallel execution on multiple processors. The architecture applies when a computational problem can be split up into a number of independent sub-problems. These independent sub-problems are referred to as *tasks* in the following. The process architecture of a Supervisor-Worker program is depicted in Figure 2.

Fig. 2. Supervisor-Worker process architecture

Supervisor and worker processes interact by a connector that we refer to, for the moment, as a “bag”. The supervisor process is responsible for generating an initial set of tasks and placing them in the bag. Additionally, the supervisor collects results from the bag and determines when the computation has finished. Each worker repetitively takes a task from the bag, computes the result for that task, and places the result in the bag. This process is repeated until the supervisor signals that the computation has finished. We can use any number of worker processes in the Supervisor-Worker architecture. First, we examine an interaction mechanism suitable for implementing the *bag* connector.

2.2 Linda Tuple Space

Linda is the collective name given by Carriero and Gelernter [14] for a set of primitive operations used to access a data structure called a *tuple space*. A tuple space is a shared associative memory consisting of a collection of tagged data records called tuples. Each data tuple in a tuple space has the form:

$(\text{"tag"}, \text{value}_1, \dots, \text{value}_n)$

The tag is a literal string used to distinguish between tuples representing different classes of data. value_i are zero or more data values – integers, floats and so on.

There are three basic Linda operations for manipulating data tuples: **out**, **in** and **rd**. A process deposits a tuple in a tuple space using:

out ("tag", $\text{expr}_1, \dots, \text{expr}_n$)

Execution of **out** completes when the expressions have been evaluated and the resulting tuple has been deposited in the tuple space. A process removes a tuple from tuple space by executing:

in ("tag", $\text{field}_1, \dots, \text{field}_n$)

Each field_i is either an expression or a formal parameter of the form $?var$ where var is a local variable in the executing process. The arguments to **in** are called a template; the process executing **in** blocks until the tuple space contains a tuple that matches the template and then removes it. A template matches a data tuple in the following circumstances: if the tags are identical, the template and tuple have the same number of fields, the expressions in the template are equal to the corresponding values in the tuple, and the variables in the template have the same type as the corresponding values in the tuple. When the matching tuple is removed from the tuple space, the formal parameters in the template are assigned the corresponding values from the tuple.

The third basic operation is **rd**, which functions in exactly the same way as **in** except that the tuple matching the template is not removed from the tuple space. The operation is used to examine the contents of a tuple space without modifying it. Linda also provides non-blocking versions of **in** and **rd** called **inp** and **rdp** which return true if a matching tuple is found and return false otherwise. Linda has a sixth operation called **eval** that creates an active or process tuple. It is not used in the following example.

FSP Notation

The behavioural specification for the tuple space involves describing it in the FSP process algebra-based notation [5,11,12,13]. This notation is used as a concise way of describing the Labelled Transition System (LTS) of the tuple space for analysis purposes. It is an "ASCII" notation to simplify parsing by the analysis tools. The original intention was to provide a graphical tool for drawing LTS diagrams. However, it soon became clear that this was clumsy and inappropriate for all but the

simplest of models. Hence we rather provide a means for translating and displaying FSP specifications as LTS diagrams as feedback to the designers.

Primitive components are defined as finite state processes in FSP using action prefix " $\rightarrow$ ", choice " $|$ ", and recursion. If x is an action and P a process then $(x \rightarrow P)$ 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 $(x \rightarrow P | y \rightarrow Q)$ describes a process which initially engages in either of the actions x or y , and the subsequent behaviour is described by P or Q , respectively. Guards can be used to control the choice of action.

Processes can be composed using the parallel composition operator " $||$ ". Processes interact by synchronising on their shared actions, with interleaving of all other actions. We have adopted the broadcast semantics of CSP [15] for interaction as this facilitates the composition of property automata to check that the composed system satisfies desired properties.

Tuple Space Model

Our modeling approach requires that we construct finite state models. Consequently, we must model a tuple space with a finite set of tuple values. In addition, since a tuple space can contain more than one tuple with the same value, we must fix the number of copies of each value that are allowed. We define this number to be the constant N and the allowed values to be the set *Tuples*.

```
const N = ...
set Tuples = {...}
```

The precise definition of N and *Tuples* depends on the context in which we use the tuple space model. Each tuple value is modelled by an *FSP* label of the form $tag.val_1 \dots val_n$. We define a process to manage each tuple value and the tuple space is then modelled by the parallel composition of these processes:

```
const False = 0
const True  = 1
range Bool  = False..True

TUPLE(T='any') = TUPLE[0],
TUPLE[i:0..N]
  = (out[T]
    | when (i>0) in[T]
    | when (i>0) inp[True][T]
    | when (i==0) inp[False][T]
    | when (i>0) rd[T]
    | rd[i>0][T]
    ) .
    -> TUPLE[i+1]
    -> TUPLE[i-1]
    -> TUPLE[i-1]
    -> TUPLE[i]
    -> TUPLE[i]
    -> TUPLE[i]

|| TUPLES = forall [t:Tuples] TUPLE(t).
```

The LTS for TUPLE value any with $N=2$ is depicted in figure 3. Note that we have not specified a guard for the **out** action. Hence, exceeding the capacity by performing more than two **out** operations leads to an ERROR. This is indicated by the ERROR state -1 in figure 3. This is a form of trap state [8] which, if reachable, indicates that that error is possible in the system. To aid the specification process, the LTSA compiler automatically maps such undefined transitions to the ERROR state. As shown later, the ERROR state is also used in property automata to check for the violation of safety properties.

A tuple space is then defined as the parallel composition of tuples, for all types of tuple in the `Tuples` set.

Fig. 3. TUPLE LTS

An example of a conditional operation on the tuple space would be:

```
inp [b:Bool] [t:Tuples]
```

The value of the local variable `t` is only valid when `b` is true. Each TUPLE process has in its alphabet the operations on one specific tuple value. The alphabet of TUPLESPEACE is defined by the set `TupleAlpha`:

```
set TupleAlpha
  = { {in, out, rd, rdp [Bool] , inp [Bool] } . Tuples }
```

A process that shares access to the tuple space must include all the actions of this set in its alphabet.

Animation

It is often the case that the LTS of a specified process is too complex to rely merely on inspection to convince oneself that it models the required behaviour. Animation can be used to test a specification. For instance, for the single tuple defined by the set,

```
set Tuples = {any}
```

LTSA permits a designer to step through the scenario given in figure 4. The actions eligible at any time are indicated by a tick, and the trace of actions is given on the left. However, for exhaustive property checking, we use property automata.

Fig. 4. Animation of the TUPLESPACE for tuple any

Property Automata

Checks can be made that the model satisfies certain safety properties by specifying these properties as automata and composing them with the system. For example, the following property asserts that an `in` action must always have been preceded by a matching `out` action.

```
property CHECK(T='any') = CHECK[0],
CHECK[i:0..N]
= (when (i<N) out[T] -> CHECK[i+1]
   | when (i>0) in[T] -> CHECK[i-1]
  ) .
```

This generates the image automata with the LTS shown in figure 5. As illustrated, property automata are automatically made complete by replacing any undefined

transition with a transition to the `ERROR` state. In the final system, safety property violations are identified by the reachability of the `ERROR` state.

Fig. 5. Property CHECK

This can be composed with `TUPLES`SPACE as follows using the parallel composition operator.

```

| | TUPLES =
    forall [t:Tuples] (TUPLE(t) | | CHECK(t)).

```

The LTSA analysis tool detects the following violation of property CHECK for tuples any, as well as the violation specified in the `TUPLE` itself.

```

Composing
  property CHECK(any) violation.
  property TUPLE(any) violation.
States Composed: 6 Transitions: 24 in 0ms
Trace to property violation in TUPLE(any):
  out.any
  out.any
  out.any

```

Hence, as expected, the violation in which an `in` action is executed before an `out` action is not permitted by the `TUPLES`SPACE.

2.3 Supervisor-Worker Model

We model a simple supervisor-worker system in which the supervisor initially outputs a set of tasks to the tuple space and then collects results. Each worker repetitively gets a task and computes the result. The algorithms for the supervisor and each worker process are sketched below:

```

Supervisor::
  forall tasks:- out("task",...)

```

```

forall results: in("result",...)
out("stop")

```

Worker::

```

while not rdp("stop") do
    in("task",...)
    compute result
    out("result",...)

```

To terminate the program, the supervisor outputs a tuple with the tag “stop” when it has collected all the results it requires. Workers run until they read this tuple. The set of tuple values and the maximum number of copies of each value are defined for the model as:

```

const N      = 2
set   Tuples = {task,result,stop}

```

The supervisor outputs N tasks to the tuple space, collects N results and then outputs the “stop” tuple and terminates.

```

SUPERVISOR    = TASK[1] ,
TASK[i:1..N]  =
    (out.task  ->
        if i<N then TASK[i+1] else RESULT[1]) ,
RESULT[i:1..N] =
    (in.result ->
        if i<N then RESULT[i+1] else FINISH) ,
FINISH        =
    (out.stop  -> end -> STOP) + TupleAlpha.

```

As illustrated, FSP supports the definition of conditional processes using **if then else**. The STOP process is one which engages in no further actions. For ease of use, the alphabet for a process is defined implicitly by the actions in its definition. This is generally more convenient than explicit definition. However, in order to ensure that no free actions can occur in the tuple space, we use “+” to explicitly extend the alphabet of the supervisor to include all the actions in the shared tuple space.

The worker checks for the “stop” tuple before getting a task and outputting the result. The worker terminates when it reads “stop” successfully.

```

WORKER =
    (rdp[b:Bool].stop->
        if (!b) then
            (in.task -> out.result -> WORKER)
        else
            (end -> STOP)
    )+TupleAlpha.

```

The LTS for both SUPERVISOR and WORKER with $N=2$ is depicted in figure 6.

Fig. 6. SUPERVISOR and WORKER LTS

In order to avoid detecting a deadlock in the case of correct termination, we provide a process that can still engage in actions after the `end` action has occurred. We define an `END` process that engages in the action `ended` after the correct termination action `end` occurs.

```

END    = (end -> ENDED) ,
ENDED = (ended -> ENDED) .

```

A supervisor worker model with two workers called `redWork` and `blueWork`, which conforms to the architecture of Figure 2, can now be defined as follows:

```

| | SUPERVISOR_WORKER
= (  supervisor: SUPERVISOR
    | | {redWork, blueWork}: WORKER
    | | {supervisor, redWork, blueWork}: :TUPLESAPCE
    | | END
    ) / {end / {supervisor, redWork, blueWork}. end} .

```

We use “:” to define a named process instance. The effect is to prefix each label in the alphabet of the process by the instance name eg. `supervisor`. This also supports the definition of multiple named process instances eg. workers `redWork`, and `blueWork`. For shared resources, such as the tuple space, every transition needs to be capable of being shared with any of the supervisor or worker processes. The notation “:” indicates that every action in the tuple space becomes a choice, prefixed by each of the user processes: `supervisor`, `redWork`, and `blueWork`. Finally, relabelling “/” is used to ensure that all processes engage in the same `end` action together.

Analysis

Safety analysis of this model using *LTSA* reveals no **ERROR** violations. However, it does detect the following potential deadlock:

```
Trace to DEADLOCK:
    supervisor.out.task
    supervisor.out.task
    redWork.rdp.0.stop      -- rdp returns false
    redWork.in.task
    redWork.out.result
    supervisor.in.result
    redWork.rdp.0.stop      -- rdp returns false
    redWork.in.task
    redWork.out.result
    supervisor.in.result
    redWork.rdp.0.stop      -- rdp returns false
    supervisor.out.stop
    blueWork.rdp.1.stop     -- rdp returns true
```

This trace is for an execution in which the red worker computes the results for the two tasks put into tuple space by the supervisor. This is quite legitimate behaviour for a real system since workers can run at different speeds and take different amounts of time to start. The deadlock occurs because the supervisor only outputs the “stop” tuple *after* the red worker attempts to read it. When the red worker tries to read, the “stop” tuple has not yet been put into the tuple space, and consequently, the worker does not terminate but blocks waiting for another task. Since the supervisor has finished, no more tuples will be put into the tuple space and consequently, the worker will never terminate.

This deadlock, which can be repeated for different numbers of tasks and workers, indicates that the termination scheme we have adopted is incorrect. Although the supervisor completes the computation, workers may not terminate. It relies on a worker being able to input tuples until it reads the “stop” tuple. As the model demonstrates, this may not happen. This would be a difficult error to observe in an implementation since the program would produce the correct computational result. However, after an execution, worker processes would be blocked and consequently retain execution resources such as memory and system resources such as control blocks. Only after a number of executions might the user observe a system crash due to many hung processes. Nevertheless, this technique of using a “stop” tuple appears in an example Linda program in a standard textbook on concurrent programming!

A possible solution is for the supervisor to output a “task” tuple with a special stop value. When a worker inputs this value, it outputs it again and then terminates. Because a worker outputs the stop task before terminating, each worker will eventually input it and terminate. This termination technique appears in algorithms published by the designers of Linda [16]. The revised algorithms for supervisor and worker are sketched below:

Supervisor::

```
forall tasks:- out("task",...)
forall results:- in("result",...)
out("task",stop)
```

Worker::

```
while true do
  in("task",...)
  if value is stop then out("task",stop); exit
  compute result
  out("result",...)
```

The tuple definitions and models for supervisor and worker now becomes:

```
set   Tuples   = {task,task.stop,result}

SUPERVISOR    = TASK[1],
TASK[i:1..N]  =
  (out.task ->
    if i<N then TASK[i+1] else RESULT[1]),
RESULT[i:1..N] =
  (in.result ->
    if i<N then RESULT[i+1] else FINISH),
FINISH =
  (out.task.stop -> end -> STOP)
  + TupleAlpha.

WORKER =
  (in.task -> out.result -> WORKER
  | in.task.stop -> out.task.stop -> end -> STOP
  ) + TupleAlpha.
```

The revised model does not deadlock and does not violate any safety property.

Progress

We have found a check for a kind of liveness properties which we term *progress* to provide sufficient information on liveness in many examples[10]. Progress asserts that in any infinite execution of the system being modelled, all actions can occur infinitely often. In performing the progress check, we assume strongly fair choice, according to which if a choice is executed infinitely often, all transitions enabled are selected infinitely often. For instance, we can use the following progress property to check that our supervisor-worker model does indeed progress to action ended.

```
progress END = {ended}
```

LTSA reports: No progress violations detected.

Fig. 7. Trace of Supervisor-Worker model

On the other hand, we can ask if, say, the workers can always accept a tuple from the tuple space, i.e.

```
progress TASK = {{redWork,blueWork}.in}
```

As expected, we then get the following violation:

```
Progress violation: TASK
Trace to terminal set of states:
supervisor.out.task
supervisor.out.task
redWork.in.task
redWork.out.result
supervisor.in.result
redWork.in.task
redWork.out.result
supervisor.in.result
supervisor.out.task.stop
redWork.in.task.stop
redWork.out.task.stop
blueWork.in.task.stop
blueWork.out.task.stop
end
Actions in terminal set:
{ended}
```

This indicates that after receiving the `task.stop` tuple, the workers can no longer accept tuples, and that only the action `ended` is available in the terminal set. The sample trace again has the red worker computing both tasks. This trace can also be generated by animation as shown in Figure 7.

Minimisation

There is also a *hiding* operator `@` which captures the notion of *external interfaces* of components, and is used in the specification of both primitive and composite components. Operator `@` specifies the set of action labels (alphabet) which are visible at the interface of the component and thus may be shared with other components. It restricts the alphabet of the LTS to the actions prefixed by these labels. All other actions are “hidden” and will appear as “silent” or “`τ`” actions during analysis if they do not disappear during minimisation (minimisation is performed with respect to observational equivalence as defined by Milner [17]).

For instance, we can abstract from many of the actions of the supervisor-worker model to examine only those in actions of the workers. This can be specified as follows:

```
|| MINIMISE = SUPERVISOR_WORKER
               @ { {redWork, blueWork}.in.task,
                  end
                }.
```

Minimisation produces a system with only seven states. The LTS is shown in figure 8. This clearly illustrates the actions of the workers in accepting tasks and finally in dealing with the stop task.

Fig. 8. Minimised LTS for the Supervisor-Worker model with hiding

3. Conclusions

In this paper, we have modelled the Supervisor-Worker architecture without reference to a specific application. We were able to discover a problem with termination and to provide a general solution that can be used in any application implemented within the framework of the architecture. Thus modelling has been used as an integral part of the design process, at a fairly high level of design abstraction.

Detailed aspects of a system can also be modelled and analysed. However, 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[11] 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. The effect of using compositional reachability to reduce the state space in this example can be seen from the table below:

(Badge, Location)	Reachable States	Transitions	Memory (Mb)	Times (secs)
(2,3)	12,213	52,758	3	6
minimised	3,924	19,260	<1	12
(2,4)	58,384	252,576	13	38
minimised	13,776	69,616	4	74
(2,5)	202,275	871,350	52	180
minimised	39,600	201,650	22	337
(2,6)	566,820	2,428,488	173	400
minimised	98,316	498,600	68	1,273

We believe that analysis and design are closely inter-linked activities which should proceed hand in hand. The notation FSP 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/concurrency>.

The approach we have described in this paper to analyzing software architectures is a general one, which is not restricted to a particular tool-set. For example, CSP/FDR [15,18] has been used with the architectural description language Wright [19] and both LOTOS/CADP [20] and Promela/SPIN [21] have been used in the context of

analysing software architectures. Our approach is distinguished by the direct use of the architecture description in analysis, the use of compositional reachability as a way of managing the state space and, hopefully, by the ease of use of the toolset.

Finally, we have experience of teaching the approach to a variety of students [13]: undergraduate students in the second year of three and four year Computing, Software Engineering and combined Computing/Electrical Engineering degree courses; and graduate students taking conversion courses in Computing. We are also investigating the utility of the approach in industry. These efforts constitute our contribution to education and technology transfer, and help to confirm our belief that the approach and toolset can be learnt and used to good effect within the lifetime of mere mortals.

Acknowledgements

The authors would like to acknowledge discussions with our colleagues in the Distributed Software Engineering Group during the formulation of these ideas. In particular we would like to thank Dimitra Giannakopoulou and Shing Chi Cheung. We gratefully acknowledge the EPSRC (BEADS GR/M 24493) and the EU (C3DS ESPRIT project 24962) for their financial support.

References

- [1] Clarke, E.M., Wing, J.M., et al. *Formal Methods: State of the Art and Future Directions*. ACM Computing Surveys, Vol. 28 No. 4 (1996), 626-643.
- [2] Cleaveland, R., Smolka, S.A., et al. *Strategic Directions in Concurrency Research*. ACM Computing Surveys, Vol. 28 No. 4 (1996), 607-625.
- [3] Magee J., Dulay N., Eisenbach S., Kramer J., *Specifying Distributed Software Architectures*, Proc. of 5th European Software Engineering Conference (ESEC '95), LNCS 989, Springer-Verlag (1995), 137-153.
- [4] Magee J., Dulay N. and Kramer J., *Regis: A Constructive Development Environment for Distributed Programs*, Distributed Systems Engineering Journal, Vol. 1 No. 5, Special Issue on Configurable Distributed Systems, (1994), 304-312.
- [5] Kramer, J. and Magee, J. *Exposing the Skeleton in the Coordination Closet*. Proc. of 2nd International Conf. on Coordination Models and Languages (COORDINATION '97), LNCS 1282, Springer-Verlag (1997), 18-31.
- [6] 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 (ed. Cleaveland R. and Jackson D.), Vol. 6 No. 1, (1999).
- [7] Cheung S.C. and Kramer J., *Context Constraints for Compositional Reachability Analysis*, ACM Transactions on Software Engineering and Methodology TOSEM, Vol. 5 No.4, (1996), 334-377.
- [8] Cheung, S.C. and Kramer, J. *Checking Safety Properties using Compositional Reachability Analysis*. ACM Transactions on Software Engineering and Methodology TOSEM, to appear, (1999).
- [9] Cheung, S.C., Giannakopoulou, D., and Kramer, J. *Verification of Liveness Properties using Compositional Reachability Analysis*. Proc. of joint 6th European Software Engineering Conference (ESEC '97), 5th ACM SIGSOFT Symposium on the Foundations

- of Software Engineering (FSE '97), Zurich, Switzerland. M. Jazayeri and H. Schauer Eds, LNCS 1301, Springer Verlag (1997), 227-243.
- [10] Giannakopoulou, D., Magee, J., and Kramer, J. *Checking Progress in Concurrent Systems*, Imperial College, London, Technical Report DoC 98, (1998).
- [11] Magee, J., Kramer, J., and Giannakopoulou, D. *Analysing the Behaviour of Distributed Software Architectures: a Case Study*. 5th IEEE Workshop on Future Trends of Distributed Computing Systems, Tunisia (1997), 240-247.
- [12] Magee, J., Kramer, J., and Giannakopoulou, D. *Software Architecture Directed Behaviour Analysis*. Ninth IEEE International Workshop on Software Specification and Design (IWSSD-9), Ise-shima, Japan, (1998).
- [13] Magee, J. and Kramer, J., *Concurrency – State Models and Java Programs*. John Wiley & Sons (1999).
- [14] Carriero, N. and Gelernter, D. *Linda in Context*. Communications of the ACM Vol. 32 No. 4 (1989), 444-458.
- [15] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, N.J., (1985).
- [16] Carriero, N. and Gelernter, D. *How to write parallel programs: a guide for the perplexed*. ACM Computing Surveys Vol. 21 No. 3 (1989), 323-358.
- [17] Milner, R., *Communication and Concurrency*. Prentice-Hall (1989).
- [18] Formal Systems, *Failures Divergence Refinement: FDR 2.0 User Manual*, ed. Formal Systems (Europe), Oxford, U.K, (1996).
- [19] Allen, R. and Garlan, D. *A Formal Basis for Architectural Connection*. ACM Transactions on Software Engineering and Methodology TOSEM, Vol. 6 No.3, (1997), 213-249.
- [20] 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, LNCS 1217, Springer-Verlag (1997), 239-258
- [21] Holzmann, G.J., *The Model Checker SPIN*. IEEE Transactions on Software Engineering, Vol. 23 No. 5 (1997), 279-295.