

Behaviour Analysis based on Software Architecture

Dimitra Giannakopoulou, Jeff Kramer and Jeff Magee

{dg1,jk,jnm}@doc.ic.ac.uk

Department of Computing, Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, UK.

1. Introduction

In our approach, software architecture design has been identified as the common underlying structure of the various phases of system development. To address this requirement, the Darwin architecture description language has been designed to be sufficiently abstract to support multiple views. In this way, software architecture describes the basic structure, which can be enriched with behavioural specifications for analysis (behavioural view) and service implementation for construction (service view) (Figure 1). 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.

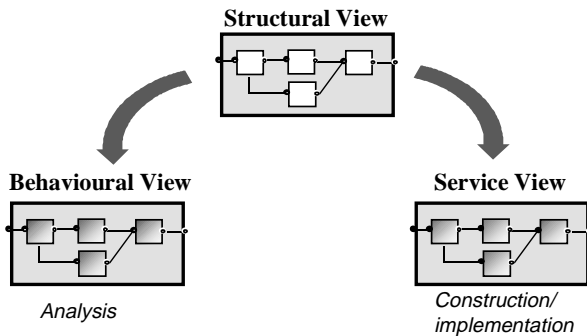


Figure 1 – Common Structural View with Service and Behavioural Views

The authors have extensively investigated the use of the architecture description language Darwin for specifying the structure of distributed systems and subsequently directing the construction of those systems [1, 2]. Our work currently concentrates on the use of Darwin as the framework for specifying the component structure and their potential interactions for the purpose of behaviour analysis rather than system construction.

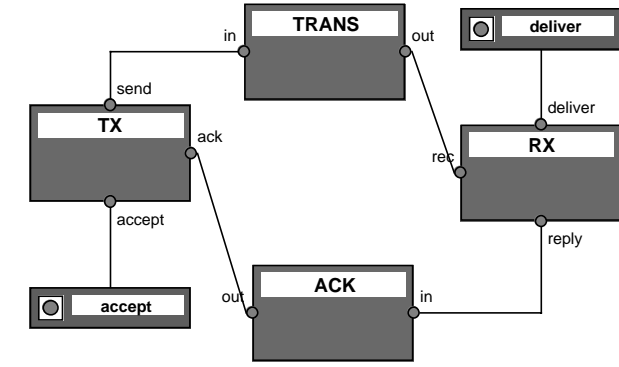
For the sake of usability, we believe that analysis and modelling should go hand in hand with design so as to

permit distributed systems designers to explore and check their designs incrementally and naturally. Our approach, Tracta, analyses behaviour described in terms of labelled transition systems (LTS). To closely match the hierarchical way system architecture is described, Tracta offers the option of performing analysis in a compositional way (compositional reachability analysis – CRA). In addition, it is equipped with techniques for checking both safety [3] and liveness [4] properties. Tracta is supported by the LTSA software tool, which provides for automatic composition, analysis, minimisation, animation and graphical display. In this paper we use the alternating bit protocol example for presenting the way Darwin descriptions can be used to direct the analysis process, as well as the analysis capabilities of our methods and tool.

2. Behaviour specification

The alternating-bit protocol is a communication protocol designed to ensure reliable transmission, despite unreliable communication lines. In our case study, the transmission medium consists of channels that may lose messages, but not duplicate or corrupt them.

The protocol consists of four components, organised in a structure described in the Darwin language as illustrated in Figure 2 (design and Darwin code produced by the Software Architect's Assistant tool [5]). TX, which is an instance of component type TRANSMITTER, tags every message sent with the bits 0 and 1 alternately (hence the name of the protocol), and these bits also constitute the acknowledgements. Therefore, every time transmitter TX accepts a new message, it initiates a new round of interactions between the components of the protocol. Every round is characterised by the bit b with which messages are tagged. When TX initiates a round where messages are tagged with bit b , it expects b as an acknowledgement - any different acknowledgement is ignored. Component instance RX of type RECEIVER works in a symmetrical fashion. Superfluous retransmissions of messages and acknowledgements during the previous round are thus ignored. Retransmissions are determined by means of timeouts.



```

component ABP {
portal
  accept: MESG;
  deliver: MESG;
inst
  TRANS : CHANNEL;
  TX : TRANSMITTER;
  RX : RECEIVER;
  ACK : CHANNEL;
bind
  TX.send -- TRANS.in;
  TRANS.out -- RX.rec;
  RX.reply -- ACK.out;
  ACK.in -- TX.accept;
  RX.deliver -- deliver;
  accept -- TX.accept;
}

```

Figure 2 – Darwin description of the alternating-bit protocol

Component instances TRANS and ACK are of type CHANNEL, and implement the lossy communication lines connecting TX and RX. Components TRANS and ACK are used for the transmission of messages and acknowledgements, respectively.

The interface of the alternating bit protocol consists of a set of actions (declared as *portals* accept and deliver) which are bound to their counterparts in the primitive component specifications.

2.1 Primitive Components

Since the content of messages is unimportant for the protocol, we only model the bits with which messages are tagged. We model the protocol for the case where the ACK and TRANS lines have capacity for at most one bit.

The behavioural specification for the protocol involves describing each of its primitive sub-components in the FSP process algebra-like notation [6, 7]. This notation is used as a concise way of describing the Labelled Transition System (LTS) of the component for analysis purposes. It is an “ASCII” notation to simplify parsing by the analysis tools.

```

range BIT = 0..1

//Transmitter starts in the state ACCEPT[0]
TRANSMITTER = ACCEPT[0],
ACCEPT[b:BIT] = (accept -> SEND[b]),
SEND[b:BIT] = (send[b] -> SENDING[b]),
SENDING[b:BIT] = (txto -> SEND[b]
| ack[b] -> ACCEPT[!b]
| ack[!b] -> SENDING[b]).

//Receiver starts in the state REPLY[1]
RECEIVER = REPLY[1],
DELIVER[b:BIT] = (deliver -> REPLY[b]),
REPLY[b:BIT] = (reply[b] -> REPLYING[b]),
REPLYING[b:BIT] = (rxto -> REPLY[b]
| trans[!b] -> DELIVER[!b]
| trans[b] -> REPLYING[b]).

```

```

// Generic lossy channel
CHANNEL = ( in[b:BIT] -> lose -> CHANNEL
| in[b:BIT] -> out[b] ->CHANNEL)
@ {in, out}.

```

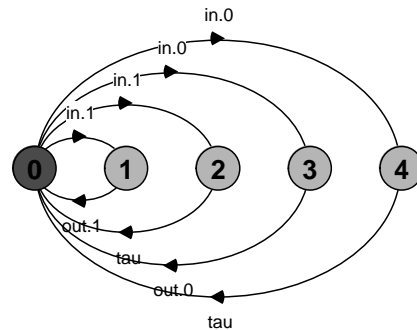


Figure 3 – Primitive component behaviour for the alternating-bit protocol

Primitive components are defined as finite state processes in FSP using action prefix “->”, 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.

The *hiding* operator $@$ captures the notion of *external interfaces* in Darwin, 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 in [8]).

Figure 3 gives the primitive component specification for component types `TRANSMITTER`, `RECEIVER`, and `CHANNEL`. The LTS for `CHANNEL` is also included in diagrammatic form. Our analysis tools automatically generate such diagrams as an aid to comprehension. According to the `TRANSMITTER` specification, a transmitter accepts messages (action `accept`), and then sends them tagged with bits 0 and 1 alternately (action `send[b]` with `b` representing the tag). Subsequently, the transmitter waits for an acknowledgement to the message sent. In that state it may: – either timeout (action `txto`) in which case it sends the message again (returns to state `SEND[b]`) – or receive an acknowledgement tagged with `b`, in which case it is ready to accept a new message which will be tagged with `!b` (state `ACCEPT[!b]`), – or receive an acknowledgement tagged with `!b` from the previous round of interaction, which it ignores. `RECEIVER` is specified in a symmetrical way. Finally, `CHANNEL` receives messages tagged with a variable in range `BIT` (`in[b:BIT]`), and non-deterministically loses them or transmits them. The non-deterministic choice is made at the time of message receipt. Its interface consists of actions `{in, out}` and therefore action `lose` is internal to it.

2.2 Composite Components

Component instantiation is modelled by creating a copy of a process where each action label contained in the process is prefixed by the instance name. In this way, each component defines a scope for the actions in its behaviour. We use notation `tx:TRANSMITTER` to denote that `TX` is an instance of component type `TRANSMITTER`. As a result, the LTS of `TX` is identical to that of `TRANSMITTER` as described in Figure 3, with the difference that all actions are prefixed with “`tx.`” (`tx.accept`, `tx.txto`, etc).

Composite components are simply the parallel composition of their constituent behaviours. Communication is achieved through synchronisation of shared actions. Composition expressions use parallel composition (`||`), re-labelling (`/`), and action hiding (`@`). The *re-labelling* operator models binding; actions that correspond to bound interfaces in Darwin are re-labelled to a common name in order to be synchronised when behaviours are composed. Re-label specifications are of the form `new-label/old-label`.

All basic Darwin features have been mapped into features of our specification language and the labelled transition systems that correspond to it. This contributes significantly to the integration of analysis with software architecture, and has been reflected in our tools. The Software Architect’s Assistant (SAA) [5] is a visual

environment for the design and development of distributed programs using Darwin architectural descriptions. The SAA generates LTS expressions based on the system architecture, which can be directly used by the analysis tool for computing and analysing the behaviour of a system based on that of its primitive components. For the alternating-bit protocol described in Darwin as illustrated in Figure 2, the SAA generates the following expression:

```
|| ABP = (trans:CHANNEL || tx:TRANSMITTER
          || rx:RECEIVER || ack:CHANNEL)
/ {deliver/rx.deliver, accept/tx.accept,
   trans.in/tx.send, rx.rec/trans.out,
   ack.in/rx.reply, ack.out/tx.ack}
@ {accept, deliver}.
```

This expression describes the component instances that the protocol consists of, and the types that correspond to them. The bindings between the components have been mapped to appropriate re-labelling in the composite expression. Finally, the external interface of the protocol consists of actions `{accept, deliver}`.

3. Behaviour Analysis

We have developed the Tracta technique for analysing the behaviour of distributed systems expressed as described in the previous section [9]. Tracta is supported by an automated tool, the LTSA [6, 7]. Tracta uses compositional reachability analysis (CRA) to perform an exhaustive search of the state space of the LTS model that corresponds to the behaviour specification. More specifically, given the software architecture of a system, and the LTS descriptions of the system primitive components, the behaviour of the system is computed for analysis in steps, as follows:

1. the behaviour of every composite component is computed from that of its sub-components with reachability analysis,
2. actions that are not part of the communication interface are hidden and the behaviour is minimised with respect to observational equivalence.

Model checking is used for checking the system behaviour against a set of properties. In general, properties are expressed either in linear temporal logic (LTL) and translated into Büchi automata, or directly as Büchi automata. We have developed an efficient technique for model checking safety properties [3]. Our generic liveness property checking mechanism is based on the automata-theoretic approach to LTL model checking [10]. According to the latter, a Büchi automaton corresponding to the negation of a property is composed with the system for verification. Our property checking mechanisms have been specifically designed for our models that focus on actions rather than states [4, 11] 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 [12]. 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.

This paper particularly describes the technique and tool support for checking safety and progress properties. Note that in our example, hiding as dictated by $\{@\{accept, deliver\}$ is applied after analysis of the protocol, in order for the tools to generate more informative counterexamples. As described in what follows, counterexamples are traces of the system that demonstrate problematic system behaviour discovered during analysis.

Deadlock

Deadlocks are identified as states with no outgoing transitions in the LTS of a system. The reachable state space for our model of the alternating-bit protocol consists of 112 states and 256 transitions. Our analysis tool has detected a potential deadlock in the resulting system, and produced the following trace of an action sequence that would lead to deadlock:

```
Trace to DEADLOCK:
<accept, trans.in.0, tx.txto,
  ack.in.1, rx.rxto>
```

In addition to model checking, our analysis tools support simulation of the system execution. More specifically, the user can animate various test scenarios and examine the effect on the individual components of the system.

By animating the trace to deadlock described above, we have found out that the deadlock is caused by the fact that channels have capacity of one. This is the situation where channels are full and have committed to transmit the message they contain but the only action that the receiver and transmitter can do is send a message to their respective channels. As described in [8], for this deadlock to be avoided channels must have infinite capacity. For the specific protocol, an infinite channel can be modelled with a channel of capacity one, which overwrites the message it contains when receiving a new one (see Figure 4).

Overwriting is not a problem in our protocol when the new message is tagged with the same bit b as the one that is currently contained in the channel (this corresponds to a message retransmission). Moreover, a message tagged with bit $!b$ can only arrive if at least a b message has successfully been transmitted. Therefore overwriting in

this case is also legal, since the protocol is designed to guarantee the successful transmission of each message at least once.

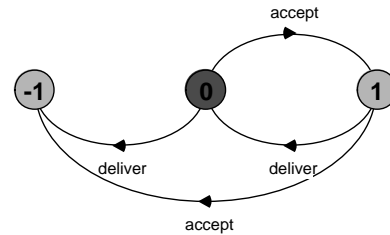
```
CHANNEL = ( in[b:BIT] -> LOSE[b]
            | in[b:BIT] -> TRANSMIT[b]),
LOSE[b:BIT] = ( lose -> CHANNEL
               | in[i:BIT] -> LOSE[i]
               | in[i:BIT] -> TRANSMIT[i]),
TRANSMIT[b:BIT] = ( out[b] -> CHANNEL
                  | in[i:BIT] -> LOSE[i]
                  | in[i:BIT] -> TRANSMIT[i])
@{in, out}.
```

Figure 4 – Modelling an infinite channel for the protocol

With the new definition of CHNL the system consists of 112 states and 376 transitions, and the analysis tool detects no deadlocks.

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 depicted in Figure 5 together with the automata it generates asserts that a *deliver* must always happen between two *accept*, and vice versa.



property AD = (accept -> deliver -> AD).

Figure 5 – Property AD

State -1 represents the *ERROR* state. As illustrated in Figure 5, 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. The analysis tool has detected no violation of property AD.

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 [12]. 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. With this assumption, the progress check finds no problem with the alternating bit protocol. However, we can impose adverse

conditions to the system by applying action priority. The following system corresponds to the alternating-bit protocol where the transmission of messages by the channels are given low priority:

```

|| ABP = (trans:CHANNEL || rx:RECEIVER || tx:TRANSMITTER
          || rx:RECEIVER || ack:CHANNEL)
/ {deliver/rx.deliver, accept/tx.accept,
   trans.in/tx.send, rx.rec/trans.out,
   ack.in/rx.reply, ack.out/tx.ack}
>> {rx.rec, ack.out}.

```

Unsurprisingly, this causes a progress check violation, since it is now possible for the transmitter and receiver to keep timing out and re-sending messages to the channels, which keep overwriting the previous messages. The tool returns the following progress violations:

```

Progress violation for actions:
{trans.in.1, rx.rec.0, rx.rec.1, accept,
ack.out.0, ack.out.1, deliver, ack.in.0}
Path to terminal set of states:
  accept
Actions in terminal set:
{trans.in.0, tx.txto, ack.in.1, rx.rxto}

```

Our methods and tool also support more refined progress tests, as described in [12]. Finally, when only actions {accept, deliver} in the interface of ABP (no action priority involved) are kept visible and the system is minimised, the observable behaviour of the system is described by the LTS of Figure 6.

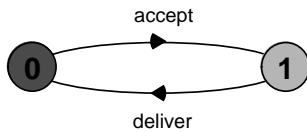


Figure 6 – Observable behaviour of ABP

4. Tools and experience

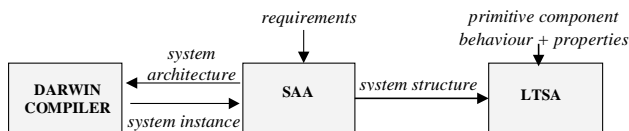


Figure 7 – Tool support for system design and analysis

Figure 7 illustrates the flow of information between the SAA tool for system design and the LTSA tool for system analysis. The basic functionality of these tools has been described in the example presented in the previous sections.

These tools have been used for applying our methods on a number of case studies. A large number of educational examples have been implemented and used in our courses for demonstrating mutual exclusion problems and other concurrency issues. Several larger case studies of

industrial protocols and systems have been analysed, most notably:

- part of a Reliable Multicast Transport Protocol designed by Lin and Paul in [13] for applications that cannot tolerate data loss. The part of the protocol analysed consists of 720 000 states and 5 724 208 transitions [9].
- an Active Badge personnel location system which has been implemented using hardware from Olivetti Research Laboratories in Cambridge. The reachable state space for a system with two badges and five locations consists of 202,275 states (871,350 transitions) [6].
- application of our dynamic change model to the case study of a Ring Database [14] based on a problem described by Roscoe [15].

5. Evaluation and Conclusion

A major objective of our work in architectural analysis has been to provide tools that are accessible and usable by practising engineers. In our approach, analysis and modelling go hand in hand with design so as to permit distributed systems designers to explore and check their designs incrementally and naturally. Additionally our tools avoid complicated notations and emphasise accessibility to non-experts. They have been developed in Java, and are available through the Web (for the SAA <http://www-dse.doc.ic.ac.uk/~kn> and for the LTSA <http://www-dse.doc.ic.ac.uk/~jnm>). The tight integration of analysis with software architecture design greatly simplifies modelling and reusability of specifications, since all changes related to the system structure are automatically performed based on the architecture.

As any exhaustive analysis method, our approach is susceptible to scalability problems. Although CRA often reduces the number of system states, it requires the storage of intermediate graphs, which can be avoided with on-the-fly methods [16]. Moreover, minimisation performed at intermediate phases of analysis is an expensive operation. For this reason, our analysis tools are extended towards supporting both compositional and on-the-fly techniques.

6. References

- [1] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. (95). *Specifying Distributed Software Architecture*. ESEC'95.
- [2] Magee, J., Dulay, N., and Kramer, J. (94). *A Constructive Development Environment for Parallel and Distributed Programs*. International Workshop on Configurable Distributed Systems, Pittsburg, USA.
- [3] Cheung, S.C. and Kramer, J. (96). *Checking Subsystem Safety Properties in Compositional*

- Reachability Analysis*. 18th International Conference on Software Engineering, Berlin, Germany, IEEE.
- [4] Cheung, S.C., Giannakopoulou, D., and Kramer, J. (97). *Verification of Liveness Properties using Compositional Reachability Analysis*. ESEC/FSE'97, 6th European Software Engineering Conference, 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland. M. Jazayeri and H. Schauer Eds, Lecture Notes in Computer Science 1301, Springer.
- [5] Ng, K., Kramer, J., Magee, J., and Dulay, N., *A Visual Approach to Distributed Programming*, in *Tools and Environments for Parallel and Distributed Systems*, A. Zaky and T. Lewis, Editors. 96, Kluwer Academic Publishers.
- [6] Magee, J., Kramer, J., and Giannakopoulou, D. (97). *Analysing the Behaviour of Distributed Software Architectures: a Case Study*. 5th IEEE Workshop on Future Trends of Distributed Computing Systems, Tunisia.
- [7] Magee, J., Kramer, J., and Giannakopoulou, D. (98). *Software Architecture Directed Behaviour Analysis*. Ninth IEEE International Workshop on Software Specification and Design (IWSSD-9), Ise-shima, Japan.
- [8] Milner, R., *Communication and Concurrency*. 89: Prentice-Hall.
- [9] 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), 98.
- [10] Gribomont, P. and Wolper, P., *Temporal Logic*, in *From Modal Logic to Deductive Databases*, A. Thayse, Editor. 89, John Wiley and Sons.
- [11] Giannakopoulou, D. (98). Expressing and checking properties in behavioural models. Dept. of Computing, Imperial College, London, Technical Report DoC 98/2, March 1998.
- [12] Giannakopoulou, D., Magee, J., and Kramer, J. (98). Checking Progress in Concurrent Systems, Technical , *submitted for publication*.
- [13] Lin, J.C. and Paul, S. (96). *RMTP: A Reliable Multicast Transport Protocol*. IEEE INFOCOMM'96, San Francisco, California.
- [14] Kramer, J. and Magee, J. (98). *Analysing Dynamic Change in Software Architectures: A case study*. 4th IEEE International Conference on Configurable Distributed Systems, Annapolis.
- [15] Roscoe, A.W., *The Theory and Practice of Concurrency*. 98: Prentice Hall.
- [16] Holzmann, G.J., *The Model Checker SPIN*. IEEE Transactions on Software Engineering, 97. **23**(5): p. 279-295.