Predictable Dynamic Plugin Systems

R. Chatley, S. Eisenbach, J. Kramer, J. Magee, and S. Uchitel

Dept of Computing, Imperial College London,
180 Queensgate, London, SW7 2AZ, UK
{rbc,sue,jk,jnm,su2}@doc.ic.ac.uk

Abstract. To be able to build systems by composing a variety of components dynamically, adding and removing as required, is desirable. Unfortunately systems with evolving architectures are prone to behaving in a surprising manner. In this paper we show how it is possible to generate a snapshot of the structure of a running application, and how this can be combined with behavioural specifications for components to check compatability and adherence to system properties. By modelling both the structure and the behaviour, before altering an existing system, we show how dynamic compositional systems may be put together in a predictable manner.

1 Introduction

There is a growing need for software systems to be extensible, as changes in requirements are discovered and fulfilled over time. In many cases it is inconvenient or costly to stop and restart an application in order to perform a change in configuration. By using a plugin architecture we can construct systems from combinations of components, with the architecture changing dynamically over time.

Szyperski describes components as units of composition that may be subject to composition by third parties [4]. Plugin architectures fit this description well. Plugins are components that can optionally be added to an existing system at runtime to extend its functionality. Each plugin may expose certain interfaces that it provides and requires [10]. By matching provisions to requirements, we can identify components that can be connected. By dynamically creating bindings between these components, calls can be made from a component requiring a service to another component that provides that service.

In an environment where systems can change through incremental addition and removal of components, it is desirable to be able to check for the preservation of properties as systems evolve. A group of components may be interacting correctly, but introducing a new component to the system may cause problems. Before adding new components, we would like to be able to ensure that undesirable behaviour will not occur, in order that configurations that might violate certain properties are not realised. Examples of such properties might be freedom from deadlock, liveness, or ensuring that an error state is never reached.

Our approach is to build and check a model that contains both structural and behavioural information.

The structural information consists of interfaces and bindings, which define sets of shared actions through which components can interact. However, they do not provide any information about the order in which these actions will be performed. This means that although we may be able to reason about the structure of systems of components based on this information, we are unable to reason in any way about their behaviour.

The behavioural information comes from the developer of a component, who can supply a specification of the way that component behaves (it is impossible to ascertain the programmer's intentions automatically). However, as components from different vendors can be combined in any number of different possible configurations, there is no way of writing a definitive model of how all different combinations will behave. To produce a model of the behaviour of the complete system requires composing the behavioural models for all of the components in a particular configuration in parallel, and ensuring that components are correctly synchronised where their interfaces are bound together. In this paper we show how such a model can be constructed, and hence the system's behaviour can be analysed.

As systems of plugin components can have components dynamically added (and removed) over time, and because one of the ideas of plugin components is to minimise the effort involved in configuring and administering a system, it is desirable that system models be generated and tested automatically. We show how our structural and behavioural specification techniques can be used for this, and how our tools can generate and analyse models automatically.

In the remainder of this paper we describe techniques for modelling the structure and behaviour of systems, and go on to discuss how the features of the plugin system map to the concepts used in these modelling techniques. We describe how models can be automatically generated from implemented components, present an example, and finally discuss related and future work.

2 Background

2.1 Plugin Framework

We have implemented a framework for plugin components, which we call MagicBeans, that can examine the compiled code of a Java component and automatically perform the matching and binding of interfaces at runtime [5]. The MagicBeans framework is more powerful than the plugin systems used to extend applications like web browsers, as we can handle plugins to plugins, creating arbitrarily complex configurations of components. An advantage over the plugin system used by Eclipse [13], is that we do not require that the system be restarted in order to pick up new plugins.

MagicBeans is implemented in Java, and allows a system to be composed from a set of components, each of which comprises a set of Java classes and

other resources (such as graphics files) stored in a Jar archive. The MagicBeans platform is a component in its own right, but the system starts with it already in place as a bootstrap. The platform provides some static methods that can be called from any other component.

The platform maintains lists of all of the components in the system and the bindings between them. When a new plugin is added to the system, the platform searches through the classes and interfaces present in the new component's Jar file to determine how it can be connected to the components currently in the system. For each component, the plugin manager iterates through all of the classes contained inside the Jar file, checking for interfaces implemented (provisions) and methods accepting plugins of particular types (requirements), and compares these for compatability with the provisions and requirements of the other components currently in the system. To be compatible, a provision must be a subtype of a requirement. In each case that a match is found, the class implementing the provision is instantiated and a reference to the object created is passed to the other component, creating a binding.

2.2 Modelling structure and behaviour

Software Architecture describes the gross organisation of a system in terms of its components and their interactions. The Darwin ADL [10] can be used for specifying the structure of component based and distributed systems. Darwin describes a system in terms of components that manage the implementation of services. Components provide services to and require services from other components through ports. The structure of composite components and systems is specified through bindings between the services required and provided by different component instances. Darwin has both a textual and a complementary graphical form, with appropriate tool support.

Darwin structural descriptions can be used as a framework for behavioural analysis. Darwin has been designed to be sufficiently abstract as to support multiple views, two of which are the behavioural view (for behavioural analysis) and the service view (for construction). Each view is an elaboration of the basic structural view: the skeleton upon which we hang the flesh of behavioural specification or service implementation.

Focussing on the behavioural view, we can use simple process algebra - Finite State Processes (FSP) [9] - to specify behaviour. A complete system specification can be written by using the same action names in the behavioural specification as in the Darwin service descriptions. These specifications are translated into Labelled Transition Systems (LTS) for analysis purposes. Analysis is supported by the Labelled Transition System Analyser (LTSA) tool.

3 Generating a model of the system

In our system of plugin components, the runtime plugin framework (MagicBeans) forms a middleware platform. This is responsible for initialising all of the components, matching the required and provided interfaces of the new component

against those of the other components already in the system, and creating bindings between them, in order to create an application. Any addition or removal of components has to be done via the plugin framework. The framework therefore has information about all of the components currently in the system, and how they are connected.

The framework can use this information to produce a textual specification in Darwin that gives a snapshot of the current system configuration. This can be done at runtime, based solely on the information present in the compiled code of the components and the current state of the system. There is no need for the developer to provide Darwin descriptions of each component, as these can be generated automatically from the bytecode.

3.1 Matching plugin concepts with Darwin concepts

Our plugin components comprise collections of (Java) classes and interfaces bundled together in a Jar file (which may also contain other resources such as graphics or data files). Below is the Java code for a basic filter plugin, and the corresponding Darwin description that is generated from it. The Java code follows an outline that would be the same for any plugin. The code for the class and interface would be compiled and packed into a Jar file, forming the plugin component.

For each Jar file we will have a corresponding `component` construct in Darwin. The Jar file may contain a number of class files representing interfaces. These are collections of methods that define types. We equate them with Darwin interface definitions which perform the same function.

```
Java :

        public interface Filter { public void data( String x ); }

        public class FilterImpl implements Filter {

             Filter next;

             // constructor
             public FilterImpl() {
                 PluginManager.register( this );
             }

             // implementation of Filter interface
             public void data( String x ) {
                 if ( next != null ) { next.data( x ); }
             }

             // method to be called by plugin platform
             public void pluginAdded( Filter f ) { next = f; }
         }
```

```
Darwin :

        interface Filter { data; }

        component FilterImpl {

             require next:Filter;
             provide Filter;
        }
```

Some of the classes in the Jar file may be declared as implementing certain public interfaces. These are classes that provide services that can be used by other components. The inclusion of such a class in a plugin is equivalent to declaring a Darwin component to have a provided port with the type named by the interface. Such a class may be instantiated several times, by a third party, to produce objects that provide this service. We do not have explicit names for these objects, so in Darwin we just declare the type of the provided port.

Components can use services provided by other components. When a new plugin is added to a system, the component that accepts it needs to be able to call methods provided by that plugin in order to use it. In Darwin this corresponds to a required port. In Java we need a reference to an object of a certain type in order to be able to call its methods. The mechanism by which we acquire such a reference in the plugin system is as follows.

An object registers as an observer with the plugin platform, by calling a static method `register()` in the PluginManager class. To be notified of new plugins, the object can define a number of `pluginAdded()` methods with different parameter types. When a new plugin is connected the platform picks the relevant method and calls it, passing a reference to the object from the new component that provides the service. In the body of the `pluginAdded()` method this reference is assigned to a field of the appropriate type.

We generate requires ports in the Darwin specification for any field in the Java which is assigned to within the body of one of the `pluginAdded()` methods. We name the port with the name of the field as declared in the class, `next` in the above example. Naming required ports is necessary as it is possible for a component to have more than one required port of the same type, as a component may accept multiple plugins of the same type. These could be assigned to different fields in the class, or added to an array. For example, a forking filter would forward data to two different downstream components, and so would accept, and keep references to, two plugins with the same interface.

It should be noted that it is much more difficult to extract information about the required services from a component than the provided service. We have to look for names of fields, and examine the body of the `pluginAdded()` method by processing the Java bytecode, rather than simply finding the type of the class. It is a trait of object-oriented programming that objects typically declare the methods that they provide, but not those that they use from other objects.

When constructing a system, we work at the level of components. A new Jar file is loaded to add a component. Any provided ports in the new component that match required ports in other components, or vice versa, are identified. The class that provides the service is instantiated by the plugin platform and any observers in the component requiring the service are notified, passing a reference to this new object. This process creates a binding between the two components. In Darwin terms, we model the complete system as a component, and add to it an instance of the providing component, which is given an arbitrary, but unique, name. We also add a binding between the relevant ports and components. The following would be generated for a chain of two filters:

```
component System {

        inst  f:FilterImpl;
              f2:FilterImpl;

        bind f.next -- f2.Filter;
}
```

3.2 Specifying behaviour

A simple example showing how these concepts might be extended to include behaviour (a specification of the order in which actions are performed) is a client connected to an email server. The Client component contains an interface Email, declaring the methods `login()`, `fetchMail()` and `sendMail()`, and when notified of an object of this type will call these methods. The Server component contains a class that implements the `Email` interface. The plugin framework can create a description of the interface and the two components in Darwin. Provided and required ports are declared with the appropriate types. In the example, the system as a whole comprises one instance each of the Client and Server components, with the two ports connected by a binding.

```
interface Email { login; fetchMail; sendMail; }

component Server {

     provide Email;
}

component Client {

     require serv:Email;
}

component System {

     inst s:Server;
          c:Client;
```

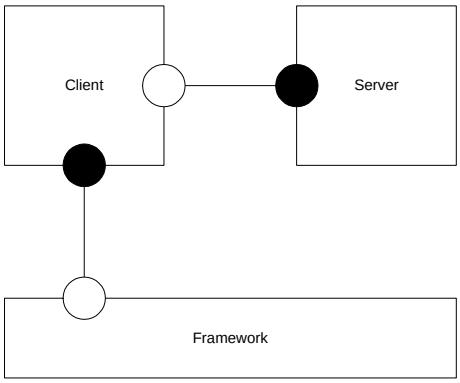

Fig. 1. Client provides FSPDefinition to the plugin framework

```
    bind c.serv -- s.Email;
}
```

The information in the Darwin description is purely structural. In order to add some information about the behaviour of each of the components, we need a way of including an abstract description of the behaviour with the component. We do not want to provide the behavioural model separately from the component as one of the ideas underpinning plugin technologies is that plugins should be deployed as single entities that include everything they need in order to be used.

The approach we have taken is to allow each component to have a another provides port where it can provide a textual description of its behaviour (in FSP) as a string. A binding can be made between this port and a requires port on the plugin framework (which is itself a component that can be connected in the same way that any other in the system can), see Figure 1.

When the framework is constructing the Darwin to describe the current state of the system, it will request the FSP from any components that provide it, and include this in the model. For example, we can include the following (Java) class in the Client component, allowing it to provide an FSP description of its behaviour:

```
public class ClientBehaviour implements FSPDefinition {

    public String getFSP() {

        return ‘‘Client = ( serv.login -> serv.fetchMail
                          -> serv.sendMail -> Client ).’’;
    }
}
```

When the framework generates the system description, it requests the FSP description from the Client and includes it in the Darwin inside the definition

of the Client component (inside a special type of comment `/% ... %/`) in the Darwin specification, as below. The behavioural description shows an ordering of actions called through the `serv` port (the client logs in, then fetches email, then sends email), which cannot be derived from the interface descriptions alone.

In the case that a component does not provide an FSP description of its behaviour, as with the Server component in this example, we generate a process that allows any of the actions from the component's provided interfaces to be performed in any order.

```
interface Email { login; fetchMail; sendMail; }
interface FSPDefinition { getFSP; }

component Client {

     require serv:Email;
     provide FSPDefinition;

     /% Client = ( serv.login -> serv.fetchMail
                   -> serv.sendMail -> Client ). %/
}

component Server {

     provide Email;

     /% Server = ( { login, fetchMail, sendMail } -> Server ). %/
}

component Framework {

     require fsp:FSPDefinition;
}

component System {

     inst s:Server;
          c:Client;

     bind c.serv -- s.Email;
          f.fsp  -- c.FSPDefinition;
}
```

Here we have included the Framework component in the model, and show how the Client provides the behavioural definition through a port which is bound to the corresponding port in the Framework. We have omitted any description of the behaviour of the Framework, as it is part of the infrastructure rather than a user component. We assume that the Framework is transparent and will not introduce any behavioural problems.

3.3 Changes of configuration

As the configuration of a piece of software constructed from plugin components changes over time, the way that particular components behave may also change. Components may behave differently depending on whether they have other components connected to their required ports.

When a plugin is connected to the system, other components need to change their behaviour to take advantage of the new services provided. Existing components need to be notified that a new component has been connected. To achieve this, components register with the plugin framework as observers, to be notified when a change in configuration occurs that is relevant to them.

The framework calls the observer back through the `pluginAdded()` method. In the FSP we can use the corresponding action `pluginAdded` action as a signal to change from one mode of behaviour to another. If a component implemented a basic matrix analysis algorithm, but allowed a plugin to be connected that provided a more efficient implementation of this algorithm, the component might perform the calculation itself while its requires port is unbound. If and when it is notified that a plugin has been added (the port has been bound), the component will change its behaviour so that from then on the call is delegated to the plugin. This could be described in FSP as follows:

```
component MatrixSolver {

    require fast:Algorithm;

    /%
    MatrixSolver = ( input -> calculate -> output -> MatrixSolver
                   | pluginAdded -> FastSolve ),
    FastSolve    = ( input -> fast.solve -> output -> FastSolve ).
    %/
}
```

3.4 Specifying properties

In FSP, safety and liveness properties can be specified for a model, and we can check these using a model checker. We also have the facility for expressing properties in a linear temporal logic. Currently we manually specify properties textually in the tool, but we anticipate that properties could be provided in components in the same way that behavioural specifications are. Properties could then either be provided as plugins in their own right, to plug in to the platform, or be integrated into other components. The platform could then incorporate them into the model.

3.5 Composing the system

The Darwin compiler constructs a parallel composition of the behaviours of each of the separate components, employing an appropriate relabelling such that

components that are bound together are synchronised. For every pair of ports that are bound, `providesport.action` is relabelled to `requiresport.action`. Any action included in a behavioural description that is not part of one of the interfaces of one the ports of a component is treated as an internal action. The resulting FSP model can be compiled to a labelled transition system and checked for properties such as deadlock [9].

When a new component is identified for addition to the system, we can determine how we intend to bind the new component, and then build a model of how the system would behave if the new component were connected in that way. If we do this before the component is connected, we can use the model to check whether adding the component will cause the system to violate any properties that we wish to hold. This information can be used to decide whether or not a new component should be bound to the system in a certain way, or added at all.

4 Predicting behaviour

We consider an example based loosely on the Compressing Proxy Problem [7]. A set of components are chained together to form a pipeline through which data can flow. We will allow further components to be plugged in to the end of the pipeline increasing the length of the chain over time. The basic premise of the problem is that in order to increase the efficiency of data transfer along the pipeline, a compression module is introduced at either end, compressing the datastream at the source and decompressing it again at the sink.

In the original Compressing Proxy Problem, the pipeline comprises a set of filters which all run in a single UNIX process. Integrating a compression module that uses gzip with this system requires some thought, as gzip runs in a separate process. An adapter is therefore used to coordinate the components. In this example we consider only the source end of this situation, although similar issues are involved at the sink.

Each component is implemented as a set of Java classes and interfaces packaged into a Jar file. We start the plugin framework, with a Source as the component that forms the core of the system. The Source generates data and sends it down stream, see Figure 2. Each component provides an FSP definition of its behaviour. The Source component includes the following class:

```
public class SourceBehaviour implements FSPDefinition {

     public String getFSP() {

          return ‘‘Source = ( next.data -> Source ).’’;
     }
}
```

Each of the other component Jar files contains a similar class. We add a plain Filter to the pipeline. The `Filter` simply reads data from upstream and passes it on downstream. `FilterImpl` implements the `Filter` interface, and has

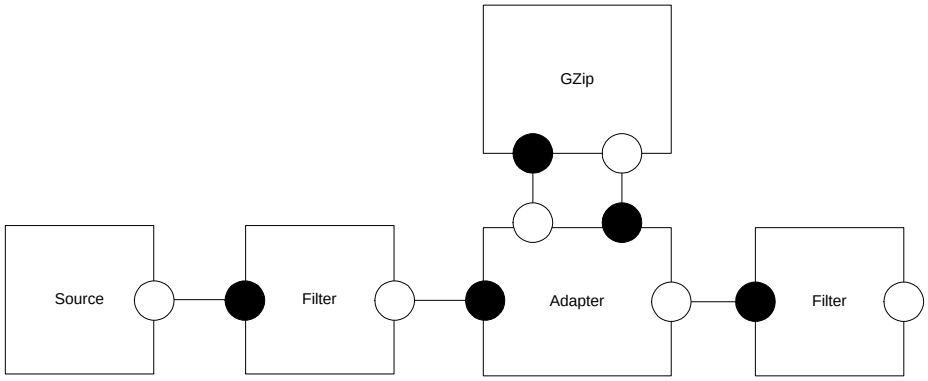

Fig. 2. Arrangement of components in pipeline with gzip

a field of type `Filter` for a reference to the next component in the pipeline. `FilterImpl`'s `data()` method just calls the next component's `data()` method. This is specified in FSP as:

```
FilterImpl = ( data -> next.data -> FilterImpl ).
```

The gzip compressor cannot be placed directly into the pipeline, and so needs an adapter component to pass data to it. The GZip component then plugs in to the adapter. When there is no gzip processor present, the adapter should behave like a plain filter. When in adapting mode, the adapter sends packets out to the processor and reads the processor's output back in before sending the processed packets on downstream. The `pluginAdded` action triggers the transition from plain filter to adapting behaviour.

```
Adapter     = FilterImpl,
FilterImpl  = ( data -> next.data -> FilterImpl
              | pluginAdded -> Adapt
              ),
Adapt       = ( data -> out.packet -> ToProc ),
ToProc      = ( out.packet -> ToProc | out.end -> FromProc ),
FromProc    = ( packet -> FromProc | end -> next.data -> Adapt ).
```

The complete Darwin and FSP specifications can be found at `http://www.doc.ic.ac.uk/~rbc/writings/fase04_appendix.pdf`. Using the Darwin compiler to translate this specification to FSP, then compiling that to an LTS model, enables the use of a model checker to perform a check for deadlock.

If we generate the model for the system without the gzip processor, then we can check the behaviour of the basic pipeline. In order to ensure that the adapter does not enter its adapting mode, it needs to be prohibited from performing the `pluginAdded` action. This can be done by composing the system in parallel with a process modelling the framework that synchronises on `a.pluginAdded` but never performs this action. Such a process can be defined as `STOP` with an alphabet extension to include the `a.pluginAdded` action.

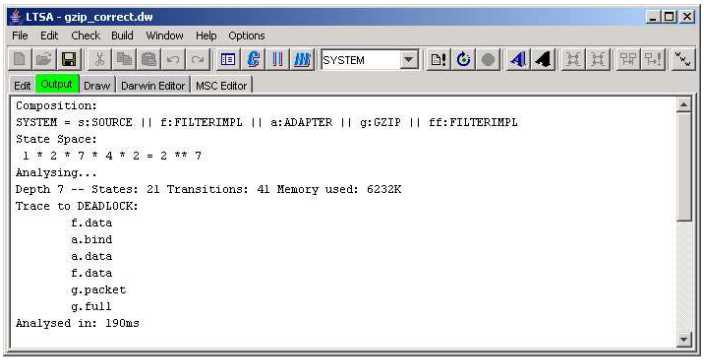

Fig. 3. Screenshot from LTSA showing trace to deadlock

```
Framework = STOP + {a.pluginAdded}.

||NoGZip = (System || Framework).
```

If we build the NoGZip process and check it, the model checker reports that it is deadlock free. If we add the GZip component, remove the constraint so that `pluginAdded` can occur, rebuild the model and check again, we find that the following trace leads to a deadlock: `f.data, a.pluginAdded, a.data, f.data, g.packet, g.full` (as shown in Figure 3). This indicates that adding the gzip processor can lead to a deadlock if GZip's output buffer becomes full before the adapter is ready to accept output from the gzip processor.

To have the system work correctly without deadlocking requires replacing the adapter component with one that will accept output from the processor before having sent it the `end` signal to say that the input has finished, or using a GZip component that never tries to write any output before it receives the end of input signal, i.e. it has infinite capacity buffers.

5 Tool support

The Labelled Transition System Analyser is a tool that compiles FSP into LTS models and checks properties on those models [9]. The LTSA itself now uses our plugin architecture, and we have developed a plugin for it to allow Darwin to be written and translated to FSP. We have also added an extension where the LTSA hosts a server that listens for Darwin specifications which are sent to it over the network.

Using these extensions, we can run a plugin application on one machine, and whenever a change is about to be made to the configuration, generate a Darwin/FSP specification and send it over the network to an instance of LTSA running on another machine. The LTSA then builds and checks the model. Producing a model of what the system would be like when a component is added before actually commiting and making the bindings, we can use the model checker

to decide whether or not adding that component and making the proposed bindings is a safe thing to do.

We could use this process to check a set of possible bindings to see if any are unacceptable because of violation of properties. Running the checks on a remote machine means that we do not have to include all of the code for the model checker in the plugin platform.

Building and checking an LTS model can be expensive in terms of computation. Depending on the frequency with which changes in the system configuration are made, and how sure the administrators need to be that the resulting system will not deadlock, it may or may not be worth doing. Checking new additions to a large business server, where changes are large, infrequent, but business critical could definitely be justified. Checking the correctness of configurations of a desktop music player application when trying out different GUI components might well not be.

6 Related Work

There is a general movement towards the idea that the specification of a component should include information about its behaviour as well as its interface [2]. Several ADLs have been extended or complemented with languages for describing behaviour, for example C2SADEL [11] which uses logic to specify behaviour, or Wright [6] and PADL [3] which use process algebra.

The idea of incorporating the specification with the component is supported by Microsoft's AsmL [1]. This allows for the runtime verification of the behaviour of the implementation against the specification.

Another angle on including within a component a way to check that it meets some property is the use of proof-carrying code [12]. Components can be provided along with a proof that they fulfil some property. The system on which they are intended to run can verify these proofs using a proof checker.

The Bandera project [14] aims to extract process models directly from Java code, so that models can be built and checked directly, without human intervention.

In the work that we have presented here, we combine the behavioural descriptions for all components and check for a property. It might be interesting to see whether it is possible to use techniques designed for finding the assumptions necessary for assume-guarantee reasoning [8] to find an assumption that must hold for a component being added to the system, and check that component against the assumption separately from the system.

7 Conclusions and Future Work

We have presented a technique for automatically generating a description of the structure and behaviour of an application that has been composed dynamically from plugin components. Using tools we can compile this description to an LTS

model, which we can test, using a model checker, to determine whether various desirable system properties hold.

The structural description can be generated automatically by the plugin middleware, based on the interfaces exported by each of the plugins and the bindings made between them. Behavioural information for each plugin is given in the form of a description in the FSP process calculus which is included in the component. By combining the behavioural information about each component with the description of the system structure, a model of the behaviour of the system as a whole can be generated.

The model can be compiled to the form of an LTS which can be analysed automatically, using a model checker, for adherence to desired system properties. Performing such analysis before a new plugin as added to the system allows us to predict whether the addition of this new component would cause the system to behave in an undesirable way.

Future work in this area could include trying to extract more behavioural information directly from the code of the components, rather than requiring the developer to write the specification by hand. Some techniques for doing this are being developed as part of the Bandera project [14] which could possibly be used. This could allow behavioural specifications to be generated automatically, rather than requiring the developer to write them in a language that may well be unfamiliar. However, if the model that is generated is too detailed then we may suffer from the state explosion problem when model-checking. Another approach would be to produce tools to assist developers in writing the behavioural specifications.

With our current technology, plugin systems are constructed by matching port types, and the techniques discussed here can be used to check the resulting system for adherence to a property. The use of behavioural properties could be extended to further direct and constrain the construction and reconfiguration of systems beyond what is currently possible.

8 Acknowledgements

We would like to acknowledge our colleagues in the Distributed Software Engineering and the SLURP group at Imperial for their participation in the discussions that helped to refine this work. We would also like to acknowledge the support of the European Union under grant STATUS (IST-2001-32298) and the EPSRC under grant READS (GR/S03270/01).

References

1. M. Barnett, W. Grieskamp, C. Kerer, W. Schulte, C. Szyperski, N. Tillmann, and A. Watson. Serious specification for composing components. In *6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, 2003.

2. M. Barnett and W. Schulte. The ABCs of specification: AsmL, behavior, and components. *Informatica*, 25(4):517–526, Nov. 2001.
3. M. Bernardo, P. Ciancarini, and L. Donatiello. Architecting families of software systems with process algebras. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(4):386–426, 2002.
4. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Pub Co, 1997.
5. R. Chatley, S. Eisenbach, and J. Magee. Painless Plugins. Technical report, Imperial College London, www.doc.ic.ac.uk/∼rbc/writings/pp.pdf, 2003.
6. D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 175–188, 1994.
7. D. Garlan, D. Kindred, and J. Wing. Interoperability: Sample Problems and Solutions. Technical report, Carnegie Mellon University, Pittsburgh, 1995.
8. J. Cobleigh, D. Giannakopoulou and C. Pasareanu. Learning Assumptions for Compostional Verification. In *Proc. of TACAS 2003*. LNCS, April 2003.
9. J. Magee and J. Kramer. *Concurrency – State Models and Java Programs*. John Wiley & Sons, 1999.
10. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Conference on Software Engineering, Sitges, Spain, 1995*, pages 137–154. Springer Verlag, 1995.
11. N. Medvidovic, D. Rosenblum, and R. Taylor. A language and environment for architecture-based software development and evolution. In *ICSE '99*, 1999.
12. G. C. Necula and P. Lee. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, Jan. 1997.
13. Object Technology International, Inc. Eclipse Platform Technical Overview. Technical report, IBM, www.eclipse.org/whitepapers/eclipse-overview.pdf, July 2001.
14. O. Tkachuk, M. B. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *Automated Software Engineering*, 2003.