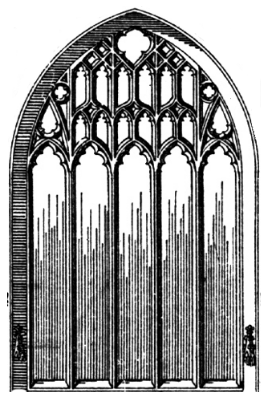
Imperial College of Science, Technology and Medicine
Department of Computing

Autonomous Architectural Assembly And Adaptation

Daniel Sykes

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College and
the Diploma of Imperial College, February 2010



Autonomous Architectural Assembly And Adaptation

Daniel Sykes
Supervised by Jeff Magee and Jeff Kramer

February 2010

Abstract

An increasingly common solution for systems which are deployed in unpredictable or dangerous environments is to provide the system with an autonomous or self-managing capability. This capability permits the software of the system to adapt to the environmental conditions encountered at runtime by deciding what changes need to be made to the system's behaviour in order to continue meeting the requirements imposed by the designer. The chief advantage of this approach comes from a reduced reliance on the brittle assumptions made at design time.

In this work, we describe mechanisms for adapting the software architecture of a system using a declarative expression of the functional requirements (derived from goals), structural constraints and preferences over the space of non-functional properties possessed by the components of the system. The declarative approach places this work in contrast to existing schemes which require more fine-grained, often procedural, specifications of how to perform adaptations. Our algorithm for assembling and re-assembling configurations chooses between solutions that meet both the functional requirements and the structural constraints by comparing the non-functional properties of the selected components against the designer's preferences between, for example, a high-performance or a highly reliable solution.

In addition to the centralised algorithm, we show how the approach can be applied to a distributed system with no central or master node that is aware of the full space of solutions. We use a *gossip protocol* as a mechanism by which peer nodes can propose what they think the component configuration is (or should be). Gossip ensures that the nodes will reach agreement on a solution, and will do so in a logarithmic number of steps. This latter property ensures the approach can scale to very large systems. Finally, the work is validated on a number of case studies.

Acknowledgements

Firstly, I should like to express my deep gratitude to my supervisors, Jeff and Jeff, for their encouragement, their humour, and their indubitable pearls of wisdom. Their support and guidance over the course of this work has been invaluable.

I would also like to thank Will and many other colleagues from DSE for their tutelage and inspiration. I count myself fortunate for having had my course swayed and checked by some wondrous mentors, not least Sophia Drossopoulou.

My sanity, insofar as it might be supposed to have survived, was by turns bolstered and besieged by the efforts of the merry band sat about me. Were it not for the entreaties to make merry—and for the provocations to engage in illuminating debate—of, chiefly, Alberto, Leonardo, Domenico and Enrico, I might have gone quite mad. The rest of the cabal must also bear their share of the blame for my pleasant state of mind.

Outside the concrete ramparts, there is a society of Mancunian expatriates who must be commended for revealing to me the other solutions in the game of life. The immeasurable and enduring friendship of Pip, Bidwell and Gemma is something I hold dear.

Finally, I must thank my mother and father, without whose care and assistance, from the dark days of the 486 DX2 unto the present, no part of this would have been possible.

Financial support was given, in part, by the Systems Engineering for Autonomous Systems (SEAS) Defence Technology Centre established by the Ministry of Defence of the United Kingdom.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

If you want to make a living flower, you don't build it physically, with tweezers, cell by cell. You grow it from the seed.

— Christopher Alexander [Ale79]

AUTONOMOUS and self-managed systems are often deployed in contexts where robustness in the face of a changing environment is required, and where their ability to reduce the effort demanded of the programmer leads to significant benefits. On the one hand, autonomous systems are useful where contact between the operator and the system is undesirable or infrequent for a reason specific to the domain. Examples include environments which are dangerous to humans, and domains where extended communication delays make close contact impractical. On the other hand, self-management reduces the burden on programmers and maintainers—providing a benefit in time and cost—by managing the inevitable change and the complexity of large-scale or highly distributed systems in a devolved manner.

Thus the advancement of autonomous systems is closely bound to the progressive devolution of responsibilities from the programmer to the system. In the simplest case, this is evident in applications which are able to control some of their own parameters in response to changes in context, using domain-specific algorithms. For example, a JIT compiler may choose to optimise different tracts of code depending on the actual behaviour at runtime, and a video streaming application may adjust the compression and resolution of the video in response to network congestion. Advanced systems allow the designer to give more general rules specifying how the system is to react under certain circumstances. In most cases, the designer explicitly states the results of adaptations within the space of possible steady-state behaviours. In other words, it is incumbent upon the designer to foresee each and every eventuality. Not only is this a complex task in itself, it is not always possible to foresee future deployment contexts.

However, a greater level of autonomy can be provided by enabling the system to respond to unpredicted changes in the environment. The response of the system must of course be one which is valuable to the user, by, for example, continuing to satisfy goals and constraints which encode the user's intent. Hence we believe a declarative, goal- and constraint-driven mechanism would be an appropriate approach to self-adaptation. Such a mechanism would, by eliding the encumbrance of explicit procedural specification of adaptive responses, enable the system to derive solutions which the designer may not have envisaged. In addition, the system would be able to adapt in response to a change of goal, something which would otherwise have required significant extra effort.

One means to perform such adaptations would be to manipulate the behaviour of the application at the language or algorithmic level, for example by adjusting numerical parameters or by applying a *mixin* to extend the behaviour of an existing class [SB98], or even by attempting to introduce a new version of a class [Dug05]. Such an approach would result in an adaptation mechanism which is specific to a particular application domain or implementation language, and which may suffer from having to deal with a multitude of peripheral issues such as preserving type safety. In addition, the technique may not scale well because of the sheer number of artefacts which have to be managed at such a low level of abstraction.

Thus we observe, as others have done before [OGT+99, OMT98], that *software architecture* fulfils the requirement for a higher level of abstraction to describe and enact adaptations which are not specific to a particular domain. Indeed, the study of self-adaptive and autonomous systems has emerged, in part, from a long line of work on software architecture and component-based software engineering (CBSE) [OGT+99]. Software architecture developed in recognition of patterns in the use of certain imprecise terms and notations, such as “blackboard” or “client-server”, to describe the design of the high-level structure of large pieces of software, independent of implementation artefacts [SG96]. The recording and formalisation of such knowledge has a didactic function, enabling the reuse of previous successful designs; eases maintenance by constricting future changes such that the overall design retains coherence; and takes computer science one step further along the road of progressive abstraction.

These benefits are achieved by treating the application as a composition of (smaller) blocks of software, called *components*, independently of who created the component, and in what language [Szy98]. In most cases, these components are explicitly marked with the services and functions that they provide and the services that they are dependent upon. Ideally, components are ignorant of their software context and only loosely coupled. *Connectors* bind provisions to requirements, making component interaction explicit, and may also encapsulate interaction-specific behaviour. Also, the physical location of components is hidden by the connector abstraction so that interaction may take place within the same host or across the expanse of the internet.

The separation of gross design structure from implementation provides a convenient abstraction in which to describe the evolution of a system over time, as the components of the system are replaced [MKM06]. In recent research, the focus has switched from static design-time evolution to evolution at runtime, moving architecture from a design discipline to a unit of adaptation within a feedback loop. Medium to large-scale adaptations can be expressed as addition, removal or replacement of components and connectors in the system (some of which may not have been available previously), largely free of domain-specific knowledge and the myriad difficulties encountered when manipulating implementation-level artefacts. One must however be aware of the state of the computation being performed by the components so that, for example, replacements can be made without disrupting the correct operation of the system.

Many self-adaptive systems use the architectural model of the application to perform adaptations at runtime [CGI$^+$08]. However, most of these systems [GS02, Che08] encode adaptations in a procedural manner not befitting our objective of a goal-driven approach. In addition, these systems assume a centralised adaptation controller with a complete model of the system architecture, disregarding the potential for components to be distributed over a network. Even when centralised manipulation of a distributed system is feasible, it suffers from a dependency on the reliability and performance of the central node, severely restricting the ability of the technique to handle large systems [IMTA05].

We take as our starting point a three-layer conceptual model (described in Chapter 3) which uses experience gained in the robotics field [Gat98] to provide a framework for developing autonomous systems. Our work is concerned with the middle layer which uses the plans generated from goals by the uppermost layer to construct and subsequently modify a (potentially distributed) configuration of components. These components implement the application behaviour and reside in the bottom layer of the model.

While we primarily aim to develop an automatic mechanism for assembling and re-assembling[1] configurations on the basis of the functional capabilities required

[1]We see no significant difference between construction and repair — both must transform the system to meet its objectives — and avoid treating the problems so differently as others have done before.

by the system's plan, it is also beneficial to consider non-functional issues such as structural constraints which limit configurations to those falling within an architectural style, or constraints over the non-functional characteristics of the components employed in the configuration [PW92]. Structural constraints provide the ability to direct the assembly process to produce solutions within an architectural style, which may reflect solutions which are known to work or may aid the user in understanding the configurations which have been generated. Structural constraints also provide a flexible way to encode certain kinds of domain-specific requirements, such as avoiding configurations which are known to be peculiarly unreliable.

Other non-functional (NF) concerns, such as performance, security, cost and reliability, can be used to guide the assembly process, if made explicit. In a system which relies on the programmer to specify the exact procedures and rules which control adaptation, NF information is used only implicitly as the programmer makes a design-time choice between components and configurations. Indeed, NF properties are often the primary reason for selecting one reconfiguration over another. For example, in a client-server system, one may wish to duplicate the server component to improve the average response time, but this must be balanced against the cost inherent in creating another server. Where NF information is explicit, it is usually used only as the impetus to invoke an adaptation procedure [GS02]. By making this information explicit within the framework of automatic assembly, the system can make those decisions previously made at design-time and can heed updated information as it becomes available.

Our work is relevant to a wide range of application areas including unmanned vehicles (robotics), embedded and mobile systems, service-oriented and client-server systems and networks. These domains all demand high (continuous) availability in the face of highly dynamic environments or rapidly evolving user requirements. However, many of our examples come from the robotics and mobile computing domains, where the interaction between the system and the physical world is particularly acute.

1.1 Requirements

The observations made hitherto lead us to the following five requirements which drive the work described in this thesis:

1. Declarative autonomy. Granting a system autonomy implies the provision of a decision-making capacity with minimal relation to a human supervisor. Turning such autonomy to the problem of architectural adaptation implies that the system must make its own choices about the best changes to apply in order to adapt, and the space of such choices is diminished the more the user sets out particular solutions. Hence, it is preferable to have the designer prescribe the minimum in relation to specific solutions, and concentrate on the requirements which a solution will uphold. Such requirements (constraints) are declarative in nature. Adapting to a change in the requirements is also eased by their explicit declaration.

2. Accounting for explicit NF properties. Whereas the designer's knowledge about non-functional properties may have been implicit in procedural approaches, here we require an explicit treatment so that the declarative adaptation mechanism can consider the impact of its choices on the NF characteristics of the resulting configuration. Where different choices are possible, this NF information can guide the adaptation mechanism to choose the most desirable configuration for the current context. This extends the problem into the realm of optimisation within the space of NF properties.

3. Enforcing explicit structural constraints. A particular case of a global non-functional constraint is that of architectural style. Since this is singularly relevant in architectural adaptation, we are interested in approaches that enforce the preservation of a style, and more generally, arbitrary structural constraints.

An unrestricted language for structural constraints also permits the encoding of domain-specific rules about the inclusion or exclusion of particular combinations of components.

4. Adaptation safety. Modifying a running system is a delicate task. While adaptation at the architectural level hides some of the difficulties, there remain complex restrictions on when and where it is safe to perform an adaptation, lest the application be jeopardised by, for example, inconsistency in the internal state of running components.

5. Decentralisation. A centralised controller places too much responsibility on a single physical host, requiring it to be reliable and provide high performance. Centralisation may also make inefficient use of the network capacity by having every node transmit its full state to the central controller. We are interested in solutions for architectural assembly that are readily distributed, providing fault tolerance and scalability.

In addition to these five hard requirements, we need to consider techniques that scale well and exhibit performance characteristics appropriate to the application domain. For example, in a robotics domain, adaptations should be computed and applied in a few seconds, since this is the time scale in which the environment can be expected not to change significantly. Other domains may be more or less strict, but by choosing an approach with better performance, a wider range of applications can be catered for.

The range of approaches which could meet these requirements is broad, and so it is necessary to make additional assumptions which lead to a particular solution. These assumptions include the existence of a repository of components which are, or can be, described in terms of provided and required interfaces, and of a mapping between functional requirements and interfaces. Moreover, we expect interfaces to form a well-structured, “clean” ontology for the domain so that names can be relied upon for matching provisions and requirements. A detailed description of our assumptions is given in Section 4.1.

1.2 Contributions

The primary contribution of this work is to provide a declarative (and largely automated) technique for adapting to changing environments and goals through assembly and re-assembly of component configurations. These configurations are derived from the abstract functional capabilities required in order to achieve some goal by a *configuration assembler*, situated in the middle layer of the three-layer model. The assembly process consists of a search within the space of configurations for those which meet both the functional requirements and additional structural constraints.

We have further developed strategies for making choices between configurations on the basis of arbitrary non-functional properties, which allow the designer to express preferences between NF properties (such as attempting to maximise performance at the cost of reliability) and hence between configurations. This is achieved by evaluating sets of functionally-equivalent components using utility functions. Rather than restricting the search space (and losing adaptation opportunities), this places solutions in a partial order so that the system can make the best choice in a given context. We also have assessed the impact of structural properties and non-functional preferences on the performance of reconfiguration, showing that it can outperform generalised constraint solving in many cases.

The second major contribution of the work is a new protocol based upon *gossip* [DGH+87] which allows the assembly process to be distributed across a network of peer nodes in order to reduce reliance on a centralised configuration assembler. Gossip removes the need for a central node that has a complete view of the system, as the peer nodes of the network come to agreement on the solution. Again, we have investigated the performance of the protocol, and found that the time taken to derive solutions is a logarithmic function of the number of nodes, allowing the approach to scale to very large systems. In addition, the protocol has redundancy so that the loss of messages or nodes can be tolerated.

In order to provide a complete approach for assembly and adaptation within the three-layer model, we have integrated the assembly algorithms with the necessary procedures for applying configuration changes. The first of these is a mechanism for preserving the application state of replaced components. The second is an implementation of the *tranquility* protocol [VEBD06b] extended to handle multiple component replacements and decentralised operation. Tranquility is a kind of change protocol used to ensure the safety of the running application while architectural changes are applied.

Chapter 3 describes the reification of the three-layer model, which was undertaken as joint work. It is not a major contribution of this thesis, although it does provide the context for the other contributions.

Much of the work has been presented in various publications including [SHMK07, SHMK08, HSMK09, SHMK10], but the constraints of publication are such that this thesis should be regarded as the definitive exposition. The first of these [SHMK07] outlines reactive planning (Chapter 3.1) and the dependency analysis of Chapter 4. The second [SHMK08] describes the algorithm of Section 4.2.3 and its application to some case studies. [HSMK09] describes the implementation of planning using the Labelled Transition Analyser and its LTL syntax (Chapter 3.1), and the incorporation of structural and non-functional information in the assembly process (Section 4.3), and includes a larger case study. Finally, [SHMK10] concentrates on non-functional selection and compares the two strategies described in Section 4.4. Each of these papers constitutes a joint work, describing one or more aspects of the work in this thesis, in addition to related topics such as planning.

1.3 Thesis Outline

In the next chapter we provide some background on software architecture and review the existing approaches for self-adaptive systems, evaluating each against our requirements.

In Chapter 3, the three-layer conceptual model for autonomous systems is described. The model stratifies the operation of an autonomous system so that the most expensive planning operations are performed the least frequently. The middle layer of the model handles architectural concerns and it is in this layer that we focus our efforts in subsequent chapters.

Chapter 4 discusses the assembly process from a centralised perspective, including the ways in which dependencies, structural constraints and non-functional (NF) information guide the selection of component configurations. Additionally this chapter makes a comparison between this dependency-guided approach and generalised constraint solving.

Chapter 5 translates the process of Chapter 4 into a fully decentralised approach using a gossip protocol, and discusses the effect on performance of factors such as the size of the network.

Chapter 6 describes the mechanisms used firstly to preserve the application state of replaced components and secondly to ensure that modifying a running configuration (to effect an adaptation) does not endanger the consistency or safety of the application. The final part of the chapter discusses implementation issues.

Chapter 7 evaluates the work against the requirements set out above and considers the efficacy of the approach in two case studies. The first of these requires several mobile robots to co-operate to achieve a global goal, and makes use of decentralised assembly. The second comes from an application for aligning satellite antennas and uses the centralised assembly process. The rest of the chapter discusses limitations that arise from our assumptions.

Finally, Chapter 8 discusses future work and concludes.

Chapter 2

Background

After briefly reviewing the context of software architecture and architectural evolution, this chapter describes the extant work on self-adaptive and autonomous systems in terms of three questions pertinent to adaptation: what must be changed, when the change should happen, and how the change is brought about in the running system. Many approaches, such as reconfiguration scripts (Section 2.3.3) and architectural planning (Section 2.3.14), have considered how to enable the designer to specify what needs to be changed in terms of the manifold combinations of connect, disconnect, create and destroy commands. Several researchers have tackled the question of when to adapt, using policies (Section 2.3.9), style violation (Section 2.3.3) or violation of non-functional requirements (Section 2.4.2). Fewer approaches have addressed how a change is enacted. The most notable of these is quiescence and its progeny (Section 2.5.3).

In reviewing the previous work on self-managing and self-adaptive systems, we refer to the criteria set out in the introduction in order to highlight the differences between the approaches. In addition, since some proposals partially achieve the requirements, we distinguish the degrees to which existing work meets requirements 1 and 2 with successive levels of increasing desirability, such as D1 to D4. These levels are given below.

1. Declarative autonomy.

The first requirement for declarative autonomy can be met partially through varying the proportions of, and relation between, procedural code and declarative statements. We define four levels, below:

- **D1 Mixed code.** The adaptive behaviour is spread throughout the normal application behaviour (or throughout the architectural description). This is difficult to understand, analyse, reuse, and modify. Avoiding this is beneficial in the same way that the development of software architecture improved the understanding of high-level artefacts which were previously implicit within the program code.
- **D2 Separate adaptation code.** The adaptive behaviour is distinguished from the steady-state application behaviour. This provides separation of concerns (easing reuse, modification, understanding) but the adaptation code remains programmatical, specifying the means to the result (in terms of connection and instantiation of specific components, and when to do so) rather than the constraints on the result.
- **D3 Independent impetus.** Separation of concerns is applied a second time to isolate the specification of the conditions or eventualities under which adaptation should be applied from the actual adaptive operations.
- **D4 Declarative adaptation.** The adaptive behaviour is independent of the application code and is specified in terms of functional or extra-functional constraints on the result rather than the means to achieve the result, necessitating some sort of search within the space of solutions. The system is now able to adapt beyond the expectations of the designer, making it more likely the system can cope with a changing environment.

2. Accounting for explicit NF properties.

The use and scope of NF information can also vary, and so we categorise them according to three levels:

- **N1 NF selection.** Non-functional properties are used to make local choices between alternative components, but solutions are not constrained by the user's requirements.
- **N2 NF matching requirements.** Non-functional properties are locally constrained, either with respect to another component's requirements or the user's requirements. For solutions within the constraints, preferences may be applied to select the best candidate.
- **N3 NF global requirements.** Non-functional properties are globally constrained by the user such that valid solutions must have some overall properties (which may not be evident from observing local information alone).

2.1 Software Architecture

Here we discuss some of the background of software architecture in order to expedite the review of previous work on architectural adaptation.

The architecture of a software system is the high-level view of the parts of which it is composed and the connections between them. The parts making up the system are called *components*. Ideally, a component should be ignorant of its context in order to maximise the opportunities for reuse [Szy98]. In a similar vein, software architecture does not prescribe the implementation of components or connections in terms of languages or protocols (though some languages allow the inclusion of such details). Thus components hide their state (data) from the architectural view. This is similar to encapsulation in object-oriented programming, though components are typically much larger and more independent than objects. Components may also contain their own threads of control, unlike objects. The architecture of a software system is the high-level view of the parts of which it is composed and the connections between them. The parts making up the system are called *components*. Ideally, a component should be ignorant of its context in order to maximise the opportunities for reuse [Szy98]. In a similar vein, software architecture does not prescribe the implementation of components or connections in terms of languages or protocols (though some languages allow the inclusion of such details). Thus components hide their state (data) from the architectural view. This is similar to encapsulation in object-oriented programming, though components are typically much larger and more independent than objects. Components may also contain their own threads of control, unlike objects.

A piece of software is created by assembling smaller components, which may have been developed independently, into larger (composite) components and eventually complete applications. We refer to such an assemblage or topology as a *configuration*.

In academia, much work has been done to provide ways to describe and specify software architecture, culminating in various *architecture description languages* (ADLs). In industry, the principles of component-based software engineering are transformed into various frameworks which support the development of applications using components. These two strands are discussed below.

2.1.1 Industrial Standards

There are several competing standards for developing component-based software, including CCM (Common Object Request Broker Architecture Component Model), EJB (Enterprise Java Beans) and COM (Component Object Model).

CORBA (the precursor to CCM), designed by the OMG (Object Management

Group[1]), defines standards for describing the interfaces of (potentially remote) objects, the protocol for invoking methods on those objects, and a central object request broker (ORB) [Szy98]. Objects providing some service define their interface using the interface definition language (IDL), and register themselves with the ORB. A client wishing to use the object can then request it from the ORB, and invoke its methods. Due to its focus on method invocation, CORBA has no explicit mechanism for expressing dependencies or hierarchical configurations. CCM attempts to mitigate this by introducing a model where components may have "facets" (provisions) and "receptacles" (requirements), and analogous ports for events [Hei05].

The OMG has also provided for architectural description within the Unified Modelling Language[2] (UML) wherein a configuration is described using a component diagram. A component in such a diagram has a number of provided and required interfaces, and may be composed of a number of other components and the connections between them.

Microsoft's COM[3] is similar to CORBA in its focus on interfaces and method calls. A COM object may provide several interfaces through which its methods can be called. Objects are initially constructed by calling a static library function with the object's "class identifier", which identifies the component in the Windows registry. A form of hierarchical composition is supported by composite components having references to the interfaces of their children, and by forwarding calls [Szy98].

Enterprise Java Beans is a standard for creating components in the Java language[4]. Each component ("bean") includes an implementation class, the interfaces it provides, a factory interface (used to instantiate the component) and a "deployment descriptor" which includes an explicit statement of the bean's dependencies. Clients are able to locate beans using the Java Naming and

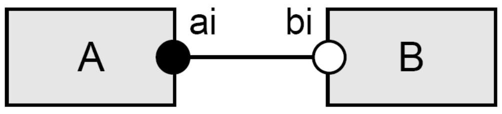
[1] `http://www.omg.org`
[2] `http://www.omg.org/uml`
[3] `http://www.microsoft.com/com`
[4] `http://java.sun.com`

Directory Interface (JNDI) [Hei05].

2.1.2 Architecture Description Languages

Architecture description languages (ADLs) allow the architecture of a system —- its high-level design —- to be expressed and analysed independently of implementation concerns. Several major ADLs and their distinctive properties are discussed here. ADLs can be roughly divided into those which associate behaviour or semantics with connectors (much like components) and those which rely on the component implementations to determine the interaction protocols.

In the latter category are Darwin [MDEK95] and Rapide [LKA+95]. A component description in Darwin states its dependencies by having a number of *ports* which either provide or require a particular interface type. A composite component contains a number of *parts* (instances of other components), and a number of *bindings* which describe the connections between the ports of the parts it contains. A composite component may also have ports which internally connect to a part. Darwin also has a graphical form, as shown in Figure 2.1. By convention, a filled circle indicates a provided port, and an empty circle indicates a required port.

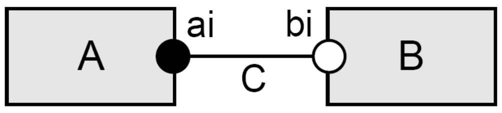


```
component A { provide ai:I; }
component B { require bi:I; }
component System
{
  inst
    a:A;
    b:B;
  bind a.ai -- b.bi;
}
```

Figure 2.1: Darwin example

Rapide is primarily focused on simulation (analysis) of the behaviour of a component-based system. Rapide's ADL features differ from the other reviewed ADLs in that a configuration is a wiring "layout" between interfaces, while actual component implementations are plugged into these interfaces. Since an architectural description in Rapide is closely bound to a behavioural description, Rapide provides a natural way to modify the architecture by creating connections when events are observed which match a certain pattern (for which purpose there is an expressive language).

The ADLs which give behaviour to connectors include Wright [AG94], UniCon [SDK$^+$95] and C2 (SADEL) [MTWJ96, TMA$^+$95]. Wright concentrates on providing a more rigorous basis for specifying architectural connection by associating with each connector and with each component port a behavioural description in a CSP-like language. Thus, connectors have an associated protocol, to which the component ports bound to the connector ports (called *roles*) must conform. Figure 2.2 shows an example configuration where the process definition on each port states that the action `i` can be performed indefinitely.



```
system s
  component A
    port ai = i -> ai
  component B
    port bi = i -> bi
  connector C
    role a = i -> a
    role b = i -> a
instances
  a: A
  b: B
  c: C
attachments
  a.ai as c.a
  b.bi as c.b
end system
```

Figure 2.2: Wright example

Later work [ADG98] extended Wright with the ability to specify changes in the architecture as the behaviour of a "configuror" component with the semantics of configuration actions given as rewrite rules. Van Eenoo *et al.* [VEHK05] further extend Wright to account for non-functional properties. These aspects are discussed in detail in subsequent sections.

UniCon is similar to Wright in that components have *players* (ports) which, when bound to a connector, play *roles* (ports of a connector). Connectors thus have structure and can be formed from compositions of more primitive connectors. In addition connectors have particular implementation semantics such as procedure call or file I/O.

C2 was originally created as an architectural style [GAO94] and developed into an ADL where components are restricted to one port on their “top” and one on their “bottom”, creating a strictly layered style (limiting the configurations it can describe). No direct connections between components are permitted. Instead communication passes through a connector, which may broadcast or filter the messages. A connector may be connected to multiple components above or below.

2.2 Architectural Evolution

The static view of architecture prevailing in the various ADLs was not likely to stand for long, given the very malleable nature of software, and since software must be changed as project requirements shift, as the technological environment develops, and under the normal process of maintenance. Before moving on to self-adaptive systems which evolve at runtime, we discuss some of the work which addresses the issues of design-time evolution, since one would expect some similarity if this design-time evolution were seen as a slow form of adaptation.

2.2.1 Backbone

Backbone [MKM06] is one such system for designing and modifying configurations of components. It provides a language (much like Darwin) and a graphical tool for expressing and then instantiating architectural designs where the leaf components are implemented as Java objects.

Backbone provides two special design constructs for specifying how a configuration changes: *resemblance* and *redefinition*. The user can take an existing composite component and add or remove parts, almost arbitrarily, to create a new composite.

The new component is then said to *resemble* the old one. Alternatively, the user can add or remove parts, and additionally use this modified design wherever the original component was used. In this case, the old component has been *redefined*. The essential difference, then, between resemblance and redefinition is that the design created by resemblance is given a new name (it is a new component), while the design created by redefinition takes on the original name (it is not a new component). Resemblances and redefinitions can be organised into strata, and so a system can be defined by applying a number of strata to a base stratum. This is somewhat similar to a mixin layer [SB98] (see Section 2.2.2), except at the architectural level.

Backbone also supports a limited form of runtime evolution by way of a *factory*, which closely resembles the direct dynamic instantiation of Darwin (see Section 2.3.1). A factory is a description of a partial configuration which is to be instantiated at runtime. This partial configuration contains a number of new components which are those to be instantiated, and a number of placeholders. When the factory is called, existing components are assigned to each of the placeholders (the placeholders act as formal parameters), and the factory instantiates the new components and connects them to the placeholder components as specified.

2.2.2 Feature Grammars

The work of Batory and others [Bat05, SB98, Bat06, BLHM02] is concerned with constructing different programs (products) from a set of features, giving a product family. This is an alternative – though compatible – way to consider the problem of architectural configuration. A feature can be encapsulated in a component, and a combination of features would thus give a configuration. In [Bat05] Batory describes the concept of a *feature grammar* which, when elaborated, enumerates all the valid configurations, and thus describes the product family. For example, the grammar

$$program ::= a|B$$

$$a ::= CD$$

states that valid configurations are the component B alone, or the pair C and D. No other combinations are valid. It may be the case that C is a component which has a required interface provided by D (or *vice versa*).

Batory shows how the grammar can be translated into propositional logic, permitting a candidate architecture to be checked easily. This reflects the fact that the work on product lines is largely methodological in relying on the user to suggest new combinations of features (new products) at design-time, thus precluding runtime evolution. Indeed, the notion of a feature implies a level of granularity which is meaningful to the user [vdS07]. No such restriction applies to components, although [vdS07] observes that certain components may represent at least the interface to certain functionality and that the complete feature can be derived from the dependencies of that “root” component.

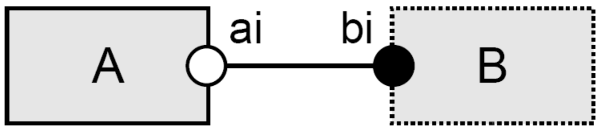
In other work, Batory describes a language-level construct, the mixin layer, for encapsulating features [SB98]. Such features have some functionality which is spread across a family of related classes, each of which must be modified (with a single mixin) to provide the overall functionality. As a non-architectural language construct, the mixin layer is restricted to design-time evolution.

2.3 Adaptive Architecture : What

In this section we review systems which are self-adaptive in the sense that they answer the first of our three questions about adaptation by deciding what the result of adaptation should be, either in a procedural or declarative manner. In subsequent sections we focus on systems which address the remaining questions of when and how an adaptation should be performed.

2.3.1 Dynamic Darwin

In an early work by Magee and Kramer [MK96], later formalised in [RE94], the Darwin ADL (Section 2.1.2) is extended with a `dyn` keyword to allow lazy instantiation of components, which enables the creation of recursive configurations of unbounded size. Whenever a component attempts to use an as-yet uninstantiated provider, a dummy provider catches the request and binds the requirement to a new provider reference. For example, in Figure 2.3, component B is only instantiated when A attempts to use it.

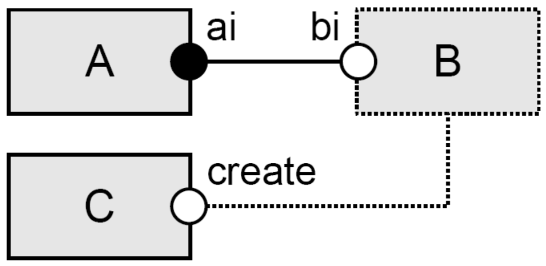


```
component A { require ai:I; }
component B { provide bi:I; }
component System
{
  inst
    a:A;
    b:dyn B;
  bind a.ai -- b.bi;
}
```

Figure 2.3: Lazy instantiation

The keyword can also annotate an interface (potentially of a third component) whose only function is to demand instantiation, as in Figure 2.4, where component C instantiates B using its `create` interface. B is then bound to the provision of A. Notice that it is redundant to have any provisions on B since any other component which depended on it would have an unsatisfied requirement r until C chose to instantiate B. Moreover, if C created multiple instances of B, then it would not be clear which of those would be called to satisfy the requirement r.

This work achieves level D2 for our first requirement, in that the dynamic behaviour is declared in the architectural description, however it cannot qualify for D3, since, by restricting dynamic instantiation to the use of a port (whether a dedicated port or otherwise), the structure of the system at any instant is determined by



```
component A { provide ai:I; }
component B { require bi:I; }
component C { require create<dyn>; }
component System
{
  inst
    a:A;
    c:C;
  bind c.create -- dyn B;
       a.ai -- B.bi;
}
```

Figure 2.4: Direct dynamic instantiation

the component implementation. In other words, the adaptation and application concerns are not separated. The architect cannot tell, for example, whether the structure will ever stop growing, without detailed analysis of the component implementations. The `dyn` keyword does not permit removal or rebinding of components, substantially restricting the forms of architectural change which can be supported.

Finally, the work does not attempt to address criteria 2 (non-functional properties), 3 (structural properties), 4 (safety) and 5 (decentralisation), though the authors allude to some of these.

2.3.2 Dynamic Wright

Early work developing Wright (see Section 2.1.2) to support reconfiguration [ADG98] recognised the need to separate the adaptive behaviour from the application behaviour. This is particularly evident in Wright due to its inclusion of behavioural specifications at the architectural level. The separation was achieved by the introduction of a "configuror" component whose behaviour was specified in terms of reconfiguration actions such as *attach* (*connect*), *detach* and *new*, which themselves had semantics in terms of rewrite rules transforming the configuration. Thus this approach can be characterised as having property D2 (separate adaptation code). However, the events initiating reconfiguration remained written within the application behaviour (at safe points only) and thus the approach cannot qualify for D3 (independent impetus), nor does it explicitly address requirement 4, that of adaptation safety. This work does have a notion of architectural style (requirement 3), but does not consider any non-functional concerns (requirement 2) nor decentralisation (requirement 5).

2.3.3 Dynamic Acme

Later work by Garlan and Schmerl [GS02] takes the separation of concerns achieved in dynamic Wright further by having a feedback loop between the system model (handling architectural concerns) and the running system (handling application concerns). The architectural style [GAO94] of the system is described in the Acme ADL [GMW00], and annotated with constraints on certain non-functional properties, such as a maximum latency. These properties are monitored in the running system. If a constraint is broken, then a repair strategy, written in an imperative language, is applied to the architectural model. The changes are then fed back to the running system. Figure 2.5 shows an example repair strategy consisting of two repair tactics, which are intended to maintain the non-functional property $p > 10$. In [CGS+02] the repair strategies are also checked for correctness with respect to the style. By separating the cause and the mechanism of adaptation from the application, this approach qualifies for D3 (independent impetus). While it does make use of non-functional information to initiate adaptation, it is left to the developer to ensure that the adaptations do improve the system. The work meets requirement 3 by using architectural style, but does not address safety (requirement 4). Finally, although Acme can manipulate systems composed of distributed components, the system model and the architecture manager responsible for executing strategies remain centralised, meaning that the work cannot meet requirement 5.

```
invariant p > 10
!-> fixP();

strategy fixP() =
{
  begin repair-transaction;
  if (fixTactic1())
  { commit repair-transaction; }
  else if (fixTactic2())
  { commit repair-transaction; }
  else
  { abort; }
}
```

```
tactic fixTactic1() : boolean =
{
  //imperative code
  return fixed;
}

tactic fixTactic2() : boolean =
{
  //more imperative code
  return fixed;
}
```

Figure 2.5: Acme repair strategy

2.3.4 Rainbow

Work on Rainbow, by Cheng *et al.* [Che08] takes the repair strategies of Acme further by calculating for each of the strategies a utility so that the most appropriate response to a constraint violation can be selected. For example, each repair strategy has a cost such as disruption of application execution, and a benefit such as a gain in performance. This removes the reliance on the designer knowing which strategies are more appropriate, by permitting the system to make choices between strategies based on contextual information which may only be available at runtime.

As in Acme, each strategy is composed of a number of tactics. However, in addition to a block of imperative code, each tactic has a condition, which states when it is useful to apply the tactic, and an *effect*, which states the expected result of applying the tactic. Tactics thus resemble event-condition-action policies [GT04]. Each tactic is annotated with the expected change in non-functional property values, in other words, a cost or a benefit. An additional non-functional property, the failure rate, is maintained by the system to record when a tactic (and hence a strategy) fails to achieve its stated effect. This property is then combined with the normal calculation of utility, allowing the system to make decisions on the basis of past behaviour.

In a strategy, tactics are arranged in a tree of condition-action rules, stating when each tactic should be applied. In order to calculate the aggregate utility of a strategy, each conditional branch of the tactic tree is given (by the user) a probability of occurring. The aggregate utility is simply the probability of each branch multiplied by the utility of the tactics in that branch. Figure 2.6 shows an example repair strategy in Rainbow intended to ensure the property p is greater than 10. Tactic branches `t0` and `t1` are labelled with probabilities 0.6 and 0.4. After taking branch `t1`, if it succeeded, then the strategy can terminate, otherwise (with probability 0.1), the system must then attempt tactic 3 (branch `t1b`).

```
strategy fixP [p <= 10]
{
  t0: (#[0.6] cond1) -> fixTactic1()
  { t0a: (success) -> done; }
  t1: (#[0.4] cond2) -> fixTactic2()
  {
    t1a: (success) -> done;
    t1b: (#[0.1] !success) -> fixTactic3()
  }
  t2: (default) -> fail;
}
```

```
tactic fixTactic1()
{
  condition { p <= 10 }
  action { /*imperative code*/ }
  effect { p > 10 }
}
```

Figure 2.6: Rainbow repair strategy

The approach relies on the assumption that several pieces of information provided by the user – the NF description of tactics, the tactic effects, the tree of probabilities – are correct. There is no mechanism for updating the information, if, for example, one particular strategy causes far more disruption at runtime than expected.

Overall, this proposal can be said to meet requirement D3 (independent impetus) since it achieves significant separation of concerns. However, although the system goes beyond the imperative nature of dynamic Acme by performing a search for the most appropriate repair strategy, it cannot be said to derive a solution from constraints as tactics remain imperative, and hence it cannot qualify for D4. The approach performs selection based on non-functional preferences (satisfying N1), and uses style as a basis (satisfying requirement 3), but does not discuss safety (failing requirement 4). Rainbow also suffers from the same centralisation as Acme, failing requirement 5.

2.3.5 Aura

In other work [GPSS04, SG02], Garlan *et al.* consider the problem of adapting in such a way as to preserve or achieve the user's task (goal). Adaptation may be the result of a change of task, a change in the environment, or the user moving between environments. They describe a three-layer architecture, where goals are managed in the top layer, configuration happens in the middle layer, and services exist in the lowest layer. Goals and services in their system are at a high level of granularity. For example, when the user's task is to edit a document, this has a direct solution by running a single text editor service. Thus, selection of services is constrained by the user's requirements, achieving criterion D4. In addition, service selection can account for non-functional preferences (N1). However, requirements 3, 4 and 5, concerning style, safety and distribution, are not addressed.

2.3.6 Distributed Management

In [GMK02] Georgiadis describes an approach to adapting systems which may be highly distributed, where there is no centralised control. This requires each component to have a manager which maintains a model of the current configuration and is charged with enforcing architectural constraints provided by the user. Managers receive notifications when the architecture changes and if at any point one of the component's requirements is not satisfied, then the manager attempts to acquire the change lock and make modifications which result in the satisfaction of the requirement while simultaneously respecting the architectural constraints. These modification scripts are written by the system architect, and in this regard the work bears great similarity to [GS02], except in a distributed setting. The problem of applying the changes while maintaining a consistent component (application) state is not addressed, and components in this system cannot be compositions. The approach does however ensure that managers have a consistent view of the global configuration through its use of reliable, totally-ordered broadcast, which restricts scalability. The approach meets criterion D3 since the adaptation concerns are wholly isolated from the application logic, but it cannot be said to be entirely declarative since the architect must still write the repair scripts. The work is not concerned with NF properties nor safety, and so it does not meet requirements 2 and 4. However, central to the approach is the notion of structural constraints, meeting requirement 3. Perhaps most importantly, the approach is completely decentralised, meeting requirement 5.

2.3.7 Genie

The approach of Bencomo *et al.* [BB09, BGF+08] is to define an adaptive system as having a set of *structural variants* (somewhat like modes, or abstractly, states) with transitions which indicate the conditions under which the variant will change. Each variant is associated with one or more configurations of components, and so transitions between variants in response to environmental change have the effect of adapting the configuration. This proposal bears some similarity to that of Zhang and Cheng (Section 2.5.1) in its use of an adaptive state machine, and also to work on policies (Section 2.3.9) since each transition can be regarded as a policy.

The work meets D3 (independent impetus) since the adaptive concerns are separated from those of the application. However, each transition and resulting configuration is specified explicitly using the OpenCOM ADL, and so D4 cannot be satisfied. The work does not address requirements 2, 3, 4 or 5.

2.3.8 Critics

Dashofy *et al.* [DvdHT02] describe an approach in which an architectural change is expressed as a “diff” (a delta). Deltas and architectural descriptions are stored in a repository, which may be augmented at runtime. Before applying the delta to the running system they apply it to a model of the architecture and use design critics [RHR96]—that is, an extensible set of analyses—to check that the result is valid. When the change has been validated it is propagated to the running system. The authors employ a simplification of the quiescence protocols described in Section 2.5.3 to ensure the adaptation process goes smoothly. First they request that the components to be removed are shut down, then that their neighbours are suspended. However, they do not account for circumstances—however pathological—in which a neighbour makes a request to one of the components to be removed after it has shut down or similarly when another neighbour makes a request to a suspended node. Indeed they state that the order of the steps in the adaptation process is irrelevant after all required suspensions have occurred. We would argue that this is not the case when we wish to minimise disruption, especially for critical components. It is clear that the proposal satisfies requirement D3 (independent impetus) and goes some way to meeting 4 (adaptation safety), but it does not meet D4 since, although the deltas are in some sense declarative, they exactly specify the resulting architecture, obviating any search for solutions. The inclusion of design critics leaves the system open to support verification of structural constraints (requirement 3), though it is not directly mentioned in the paper. The approach does not address the use of NF properties (requirement 2) nor decentralisation (requirement 5).

2.3.9 Adaptation Policies

Georgas and Taylor [GT04] describe a system where architectural change is enacted by architectural policies which are invoked in response to certain events such as component failure. Actions performed by policies may add or remove components (or connections), and even manipulate other policies. Hence, the set of policies can be extended while the system is running, but as with many other approaches, the policies are specified explicitly by the programmer. Likewise, in [GBJC07], reconfigurations are specified in the form of condition-action rules (similar to a policy), and are validated by a style enforcement service before application.

There is a wide variety of other work using policies for self-management, such as [SFLD$^+$07]. The use of policies provides a uniform mechanism for dealing with changing environments and resources, and even specification of sequential behaviour. However, excepting [GT04, GBJC07], work on policies is not explicitly architectural, and much of the work relies on the programmer writing the policies. One exception is that of Bandara *et al.* [BLMR04] on policy refinement wherein high-level "goal" policies are refined into implementable sets of policies using logical abduction, in a manner akin to planning (Section 2.3.14).

Policies allow the reason for adaptation to be specified independently (achieving D3) but, in most of the work we are aware of, policies are written by hand, and are essentially imperative. Moreover, there is no particular means for dealing with NF properties, decentralisation or safety (requirements 2, 4 and 5). [GBJC07] does deal with structural constraints (requirement 3), however.

2.3.10 Graph Grammars

Work on graph rewriting [LM98, BLM08] uses the notion of architectural style as a basis for delimiting valid transformations of the component graph in an architectural configuration. In other words, by restricting the language of transformations to style-preserving rewrites, the configuration is correct by construction, obviating an explicit constraint check.

Le Métayer [LM98] defines an architectural style as the class of configurations generated under applications of a given set of rewrite rules, such as

$$clientserver ::= client + clientserver | server$$

which states that a client-server system can be a server, or can be rewritten arbitrarily to add a client (the syntax here is simplified from that in the paper). The rewrite rules are statically checked for consistency with the intended constraints of the style, so that the rules can be applied freely at runtime.

Graph rewriting approaches naturally deal with requirement 3 (structural constraints). However, [LM98] observes that context-free grammars cannot express all possible styles. Also, these approaches do not address the question of when to apply a particular rewrite rule, nor whether it is safe to do so at a certain point in the application's computation (requirement 4). Likewise they are not concerned with decentralisation nor NF properties (requirements 2 and 5). They can be said to meet D2 since the rewrite rules are independent of the application, and can be seen as repair strategies (with all the aforementioned drawbacks) in a more abstract language.

2.3.11 Specification-Based Retrieval

Various authors [MMM97, PA99, ZW97, NR07] have proposed a component selection mechanism based on a (logical) specification of the component's functionality. Rather than binding provisions to requirements based on a syntactic match of the interface name (or method names), such matching considers how well the specification of the provision guarantees the conditions of the requirement. This kind of retrieval could be used as part of a larger adaptation scheme.

In [MMM97, PA99, ZW97] a provision is matched to a requirement if the provision's specification is a *refinement* of the requirement. For a function, this means that the post-condition of the provision must imply the post-condition of the requirement. For example, a requirement for $f(x) > 4$ would be satisfied by $f(x) > 5$. Often, though, refinement matching is too strict, and so the authors suggest various weaker alternatives. [NR07] proposes to determine the validity of a semantic match by testing. Specification-based retrieval would meet criterion D4 if combined with a larger adaptation scheme which should address requirements 2, 3, 4 and 5.

2.3.12 Service Brokering

There are a number of works in the field of web services and service-oriented architecture which are relevant for self-managed systems. Such work generally only deals with the selection of a single service to meet a requirement, but this can be seen as a degenerate case of modifying a configuration of components. For example, the approach taken by Mukhija *et al.* [MDSR07], in developing the Dino service broker, matches the functional and non-functional requirements of a service requester against the provisions of each service provider.

Matching of the functional requirement is done using a semantic description (see Section 2.3.11) in OWL-S[5]. For the matching of non-functional (QoS) requirements, each provider states its expected value for an NF property, and its confidence that the value can be achieved. Between all providers which meet the functional requirements (and the minimum NF requirements), a choice is made on the basis of evaluating utility functions.

When a binding has been made, the NF properties of the provider are monitored by the broker (requiring the broker to understand the meaning of each property), and the provider's utility is penalised (through the inclusion of a *trustworthiness* property) when the service agreement is violated, which is to say, when the provider's NF property diverges from the advertised value. This can also trigger an adaptation to select an alternative provider.

Since Dino performs a search for providers, it clearly achieves requirement D4. Moreover it accounts for NF information, satisfying requirement N2, but not requirement 4 (safety). Dino does not consider structural constraints (requirement 3),

[5]Semantic web ontology language.

reflecting the wider point that Dino is not concerned with properties, architectural or otherwise, of the system as a whole. Each selection is performed with local information, which may lead to a result which is globally sub-optimal. The Dino broker does not deal with distribution issues (requirement 5), despite the expectation that services are ordinarily distributed.

Mokhtar *et al.* [MLGI05] propose a system similar to Dino wherein web services are semantically matched against a state machine representing the user's task, and choices are made on the basis of NF requirements.

2.3.13 MADAM

The MADAM (and later MUSIC) project [BHRE07, MAD06, MUS07] considers various means for finding component configurations which have the optimal combination of non-functional properties (with respect to resource constraints), noting that the solution space is exponential in the number of variation points.

The approaches considered include exhaustive search, greedy selection and another using the Bellman-Ford algorithm [AHE$^+$06].

In particular, the MADAM approach meets D4 since it searches in a space of solutions for one which meets resource constraints, and the functional goal implied by the root component. The proposal meets (and exceeds) N3 since solutions must meet resource constraints, and the remaining NF properties must be optimised[6]. MADAM appears to support a limited form of structural constraints that encode dependencies (requirement 3), but does not meet requirements 4 and 5.

[6]The related QuA project [SE04] also considers NF properties.

2.3.14 Architectural Planning

Arshad *et al.* [AHW07, AHW04] use a planner to find an optimal method for deploying a configuration. Planning tools take a description of a system and its world [Jac95], and a specification of a goal, to produce a sequence of actions which should be performed to achieve the goal. In this case, a linear plan is produced in terms of architectural actions such as instantiation and connection. Plans are optimised in terms of the time or resources needed to deploy the configuration. The goals used may be an explicit description of the desired configuration, or an implicit description which includes only the components of interest. The use of linear plans prevents this system from being able to cope with failures during deployment. A reactive plan would be able to recover from reaching an unexpected state after performing an architectural action.

Their work also addresses adaptation in the face of component failure, by generating a new plan from the current state when a component fails. The new plan will describe how to reconstruct the configuration. The use of planning for adaptation allows the programmer to avoid having to specify the repair strategies ahead of time, as in most of the other approaches. Unfortunately, using planning for all architectural reconfiguration comes at some cost. In [AHW07] planning took between 5 and 57 seconds, while plan execution took between 62 and 138 seconds. It seems possible that a 'naïve' algorithm might perform better overall, so that it can be applied in the context of dynamic reconfiguration.

This proposal satisfies requirement D4[7], since the result is constrained rather than programmed. It also addresses the safety of the change, since architectural actions have a precondition of components being in the appropriate state, meeting requirement 4. The approach also meets N3 with the caveat that the NF properties pertain to deployment actions rather than configuration resulting from deployment. Although not explicitly mentioned by the authors, the planner's input

[7]Similarly the policy refinement of [BLMR04] meets D4.

language seems capable of handling structural constraints, which would enable the approach to satisfy requirement 3. However, the approach is centralised, failing requirement 5.

2.4 Adaptive Architecture : When

In this section we review work which considers when (and hence why) an adaptation should be performed. Much of this work pertains to reconfiguration in response to changing non-functional properties (of the environment or the system). Other causes of adaptation, such as changing goals and failing components, are described in Section 6.1.

2.4.1 Resource Allocation

Walsh *et al.* [WTKD04] use utility functions to adapt data centres in response to particular non-functional properties such as latency. Adaptation is achieved by re-allocating resources (of which there is a fixed quantity) between applications (which can be thought of as services or components) in order to maximise the utility. Each application uses the current demand on the service and future demand (predicted by a *demand forecaster*) to calculate the utility it can provide under the current resource allocation. In other words, changing demand (in the environment) can trigger the system to reallocate resources to match the demand.

2.4.2 Resource Prediction

Poladian *et al.* [PGS$^+$07] describe a mechanism for predicting resource availability, to enable pre-emptive adaptation, so that the resulting utility might be maximised. This is motivated by the observation that merely reactive adaptation (in response to changes in availability) leads to sub-optimal utility. In addition, frequent adaptation is discouraged by giving an increased utility to components that are already running.

Prediction of resource levels is performed by combining multiple basic predictors such as linear regression over the recent history, expected minima and maxima, and step changes. The set of enabled applications (which might be thought of as components) are then changed (one or more times according to different strategies) to maximise the total utility for a number of time steps into the future.

2.4.3 Failure Prediction

Epifani *et al.* [EGMT09] present an approach (KAMI) for updating NF properties provided *a priori* after monitoring their real values, in order to improve a model of the system. NF properties in this case are probabilities in a discrete time Markov chain describing the system behaviour.

Given some requirements (on reliability for example), KAMI can then use the model to detect and even predict when the system will violate its requirements. This violation could be used to adapt the system configuration pre-emptively.

2.5 Adaptive Architecture : How

In this section we review approaches for ensuring that the process of adapting from one steady-state behaviour to another is safe in the sense that it does not cause deadlocks or incorrect component execution (such as segmentation faults). Naturally, most of these techniques only address one of our requirements, that of adaptation safety (4).

2.5.1 Preserving Guarantees

Zhang, Cheng *et al.* [ZC05, ZC07, ZC06a, ZC06b] apply formal techniques to show how the safety of a transition from one steady-state program (which can be thought of as an architecture) to another can be guaranteed. In order to do this they require descriptions of the source and target programs in the form of finite state machines (which may be derived automatically [ZC07]). Each state machine is checked against its own invariant (given in linear temporal logic), and the states are annotated with their guarantees (which should include the invariant). For each transition (specified by the programmer) from a state in the source FSM to a state in the target, the transition is only valid if the guarantees of the target preserve the guarantees of the source, that is, if the target guarantees imply the source guarantees. Only the guarantees of the two states involved in the transition need be considered. In Figure 2.7 S1 and S2 belong to a source program, and SA and SB to a target program.

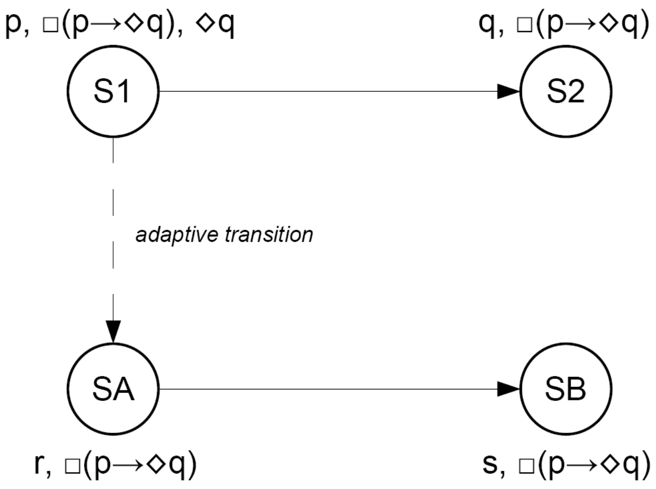


Figure 2.7: Adaptive system

The guarantees of S1, which include $\diamond q$ (eventually q), are not satisfied by the target state, since q is no longer a possibility if the adaptive transition is taken. Hence this is not a safe adaptation.

Not only does this scheme permit verification of the adaptive transition, it does it in a modular fashion. When a new source or target program is added with new transitions to existing programs, only the guarantees of the new program need to be computed. These are then easily compared against the previously computed guarantees of the other programs. Previous approaches would have to recompute all guarantees as the adaptive program was regarded as a single monolithic FSM.

This work is not directly applicable to the problem we are considering since they assume that the adaptive transitions are specified by the user (they themselves admit this requires a worst case of N^2 transitions for N programs). Indeed, this supports our claim that it is a significant burden upon the programmer to specify adaptations explicitly, and that a more declarative mechanism is needed. Also, they regard the source and target programs as monolithic entities which have no discernible internal structure. They do not tackle the problem of performing a partial change to an architectural configuration.

2.5.2 Minimising Cost

Further work by Zhang *et al.* [ZYCM04] considers safe adaptation in the context of programs composed of components. This work does not simply verify an adaptation, but derives an adaptation procedure which does not violate certain safety conditions, and which is optimal with respect to some measure of cost. The first step of their process is to determine all the valid configurations. This is done by considering the specific requirements of components (such as required interfaces), and by considering other potentially global constraints. For each possible add, remove or replace operation which transforms one configuration to another, a cost is associated. This forms a graph of configurations where the arcs are weighted. The system then finds the minimum-cost path through this graph from the source to the target architecture. For example, in the adaptation graph below, there are two ways to adapt from a configuration consisting of just A to that consisting of B and D. The path with an intermediate configuration has a lower cost and will be selected by the algorithm.

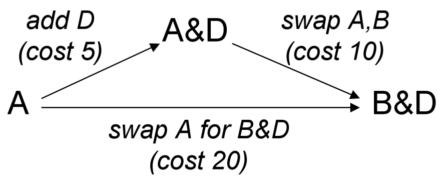


Figure 2.8: Adaptation graph

The work also includes a simple quiescence mechanism (see Section 2.5.3) whereby components are sent “reset” messages instructing them to enter a state safe for adaptation.

2.5.3 Quiescence

Kramer and Magee [KM90, KM98] describe a method to apply a previously unknown set of changes to an architecture during execution, while maintaining consistency. They consider systems in which transactions (communications between components) are independent, that is, where one transaction cannot invoke another sub-transaction before completing. Also, they restrict their scope to systems which do not have composite components – each component is a leaf.

They introduce the notion of a *quiescent* node, which is a component which is not engaged in any transaction (that it or another node initiated) and will not start any new transactions (and it will not be required to service any incoming transactions). The weaker notion of a *passive* node is one which must not be engaged in a transaction it initiated, nor initiate a new transaction (it may however service transactions). If a component is to be removed, its neighbours (those connected to it) must be passive. Components must therefore respond to a “passivate” message which ensures that eventually the required nodes will be quiescent. At this point a component can be disconnected or removed (or added). The passive set $PS(Q)$ of a node to be removed, Q, is defined to be Q and the set of nodes directly dependent on Q. The quiescent set QS is the set of all nodes which have an outbound connection which is to be removed. When all the connections to be removed are directed to a node which is to be removed, $QS = PS$. The change passive set is $CPS = \bigcup_{n \in QS} PS(n)$, denoting the set of nodes which must be made passive before the changes are performed.

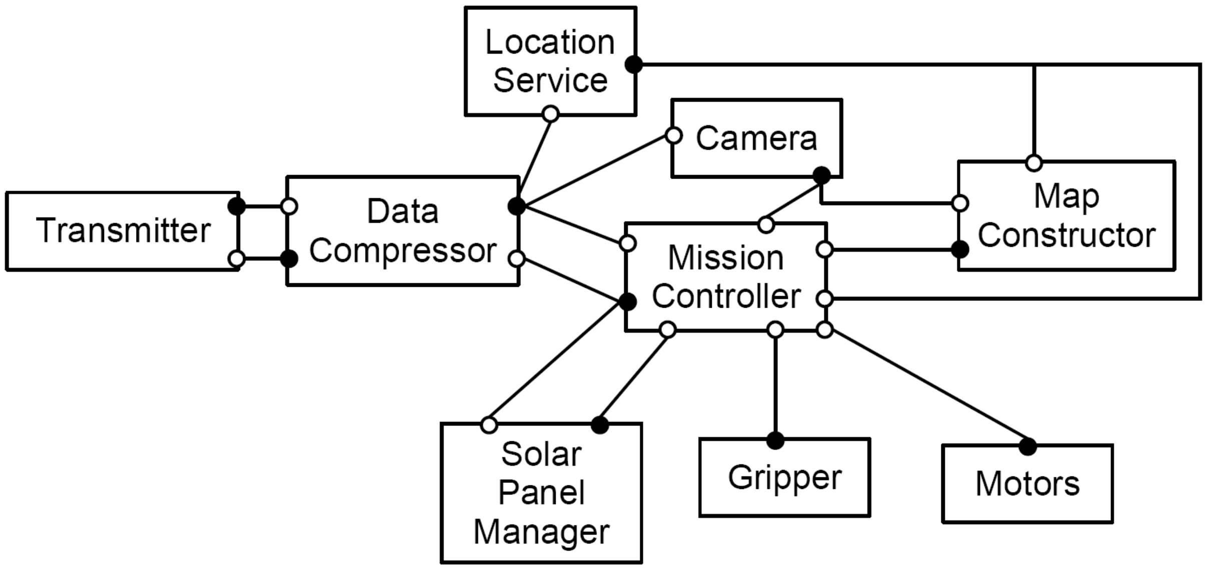


Figure 2.9: Lunar rover architecture

To demonstrate this, consider the example configuration in Figure 2.9, which describes the software architecture of a (hypothetical) lunar rover. Communication between the rover and headquarters is handled by the transmitter component. Other components use the provided interface of the data compressor to have the data compressed before transmission. Commands may come from the transmitter, and so it requires a component to handle those commands, namely the mission controller. The mission controller has direct connections to the gripper and motor components which deal with hardware details. There is also a location service which the mission controller and map constructor use to determine the current location. The purpose of the map constructor is to collect information from the rover's camera and the location service to build a map of the environment, which the mission controller may then query. The purpose of the solar panel manager is to adjust the angle of the solar panels continually so that the rover does not run out of power. This requires a connection to a mission controller, which it uses to report when power is low.

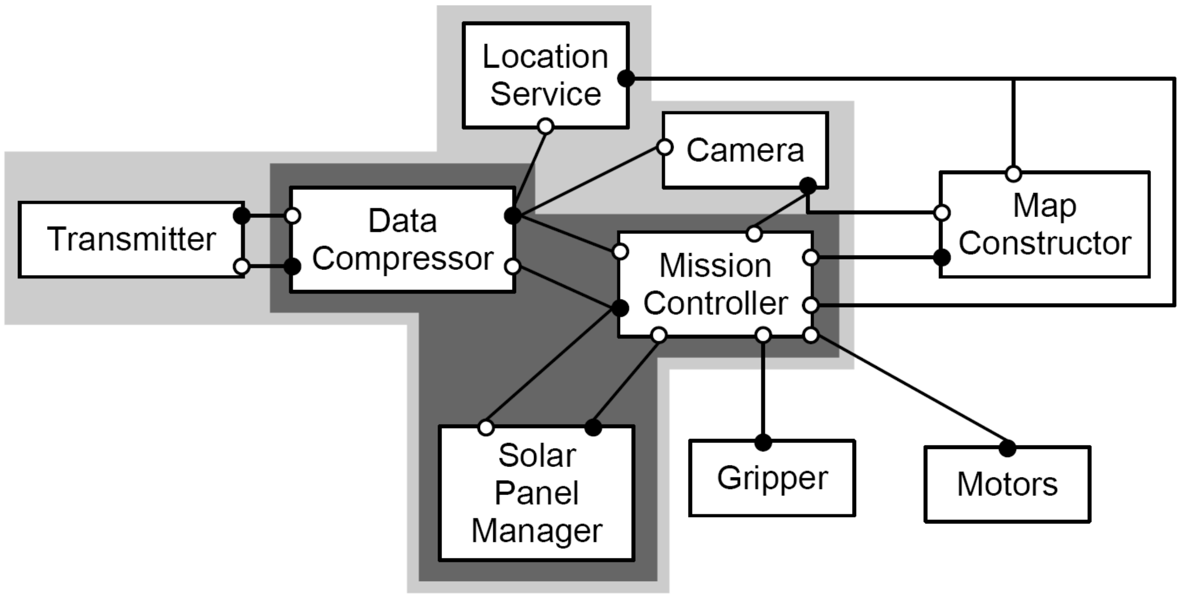


Figure 2.10: Quiescent and passive sets

If the mission controller is to be changed for an alternative implementation, then the existing one must be removed. The quiescent set $QS = \{$Mission Controller, Data Compressor, Solar Panel Manager$\}$ since this latter pair have direct connections to the mission controller. This set is shaded in Figure 2.10. The $CPS = QS \cup \{$Location Service, Camera, Transmitter$\}$, and is denoted by lighter shading.

The transmission system and the solar panel manager are critical components, and it is highly undesirable to make them passive. In particular, the solar panel manager must continue running to provide power. It is only included in the quiescent set because of a connection to the mission controller, which it uses very infrequently. This suggests a more dynamic approach that considers actual rather than potential behaviour would be more appropriate. That being said, the work is directly applicable to our problem since it permits arbitrary changes at the architectural level.

2.5.4 Dynamic Growth

Moazami-Goudarzi [MG99, MGK96] proposes a system whereby the set of quiescent nodes grows dynamically. The blocked set ($BSet$) is the set of nodes which will not initiate or service transactions (equal to QS). Once a node has been asked to block, it should continue to service requests that come from other $BSet$ members since this may be required for quiescence to be achieved. Since a request to a node outside the $BSet$ may cause a chained request to a member of the $BSet$, the $BSet$ must grow when a $BSet$ member initiates a transaction with a node previously not in the set. These extra members are later removed when the transactions terminate.

Considering the previous example where the mission controller is to be removed, $BSet = QS$. Further, suppose the mission controller calls the location service before it blocks. This causes a consequent transaction on the data compressor, which must be temporarily unblocked. This causes a further transaction with the transmitter. Hence, the location service and the transmitter are added to the $BSet$.

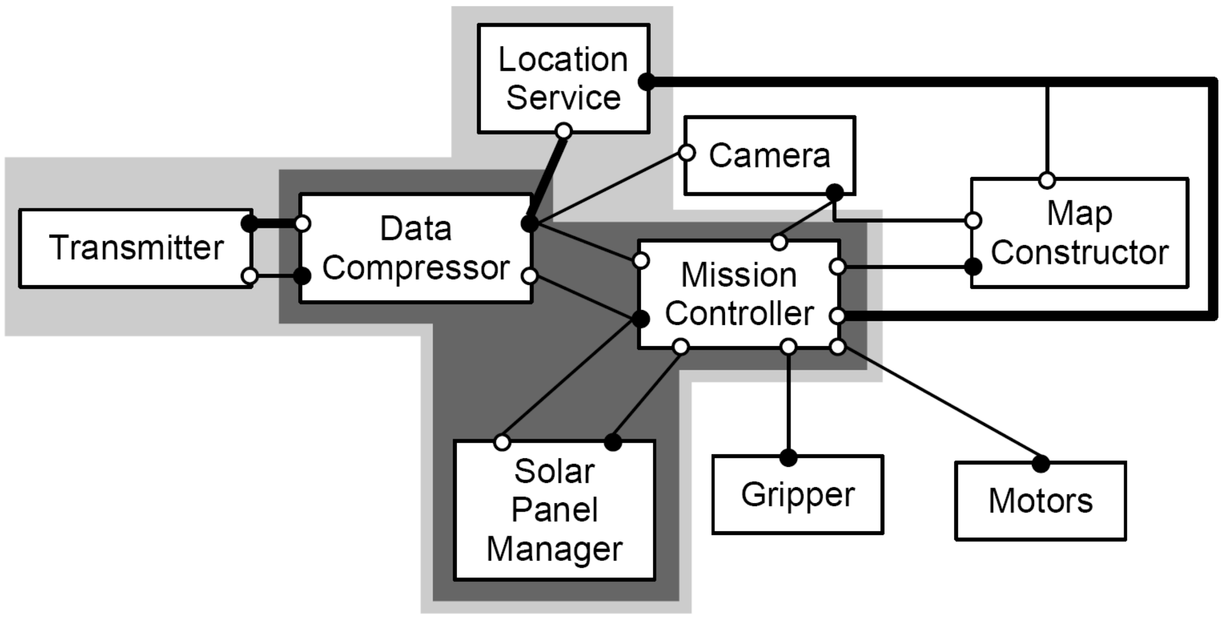


Figure 2.11: Blocked set

This is an improvement over the system in [KM90], but still requires the transmission system and solar panel manager to block.

2.5.5 Connections

Wermelinger [Wer99] presents a refinement which attempts to minimise unnecessary disruption by concentrating on blocking connections themselves rather than whole components, and only blocking connections which will be removed in the course of reconfiguration. In order to avoid deadlock, the dependency between transactions (viz. between connections) must be given, so that a dependency is always blocked after the dependant. Unfortunately, this requires substantial knowledge of component implementations.

Wermelinger also addresses hierarchical (composite) components. The difficulty is that the dependencies between connections in a composite component cannot always be seen when only considering that component, since an extra external connection from the composite component back to itself may create a dependency. Wermelinger solves this by associating a configuration manager with each component (leaf and composite). When a connection is to be blocked, the manager for the source of the connection (the dependant) is queried. This manager forwards a request to its own dependants to ask them to block the inbound connections. This chain continues to the “root” which may pass outside the current composite component. The root sends an acknowledgement to allow its outbound connection to be blocked. Acknowledgements are sent back along the chain ensuring connections are blocked in a safe order.

In the rover example, all connections to or from the mission controller will have to be blocked as in Figure 2.12.

Many of these connections are outbound, and so blocking them is not a problem. However, both the data compressor and the solar panel manager have connections towards the mission controller. It seems reasonable to expect that the controller will

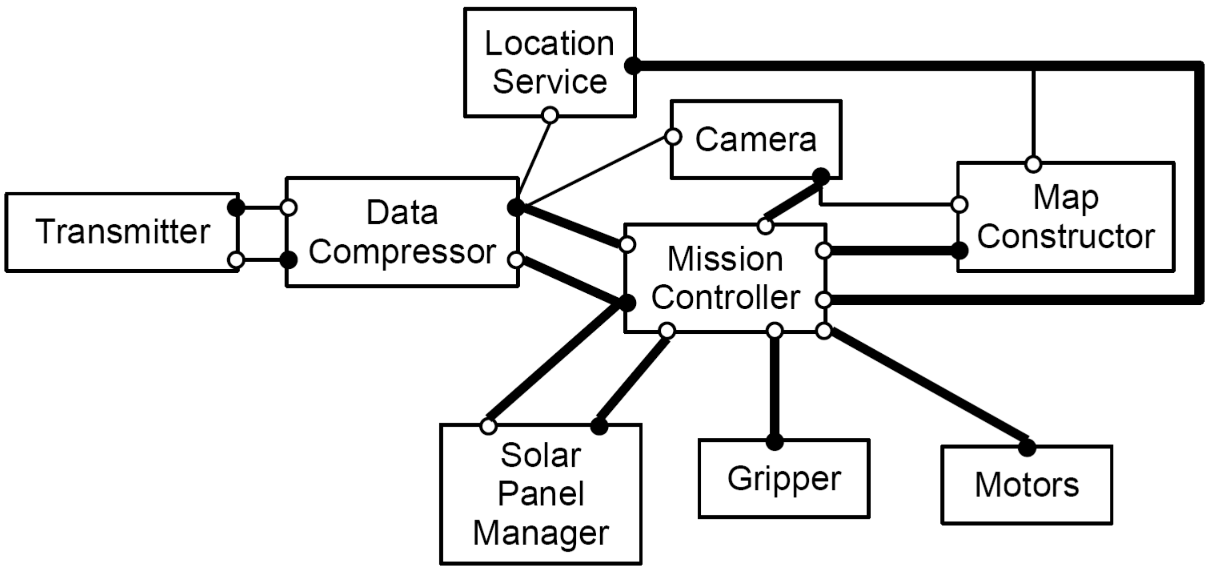


Figure 2.12: Blocked connections

be unresponsive to commands from the transmission system while being changed, and will ignore low battery events from the solar panel manager. However, low battery events are unlikely, and this scheme has the advantage of not requiring the solar panel manager to be blocked.

2.5.6 Tranquility

Vandewoude *et al.* [VEBD06a] observe that the work of Kramer and Magee causes widespread disruption and breaks the black-box principle since components must know something about their context. A new solution is proposed, called *tranquility*, which weakens the conditions of quiescence, at the cost of being unable to guarantee that it is reached in a finite time. This, though, can be managed by reverting to the original quiescence scheme after a certain period of time.

One of the conditions of quiescence is that the node to be replaced will not participate in any transactions yet to start, or currently in progress, which are initiated by another node. This is weakened to requiring that the node will not participate in any transaction in which it has already participated. To illustrate the point, consider Figure 2.13, which shows the participation of component C in a transaction T, where time flows from left to right. Both situations are prohibited by quiescence, but replacement of C is permitted under tranquility in situation (b), after C's participation has ended. Intuitively, if C has no further work in the transaction, or has not yet worked in the transaction, it can be replaced (assuming replacement finishes before C is required).

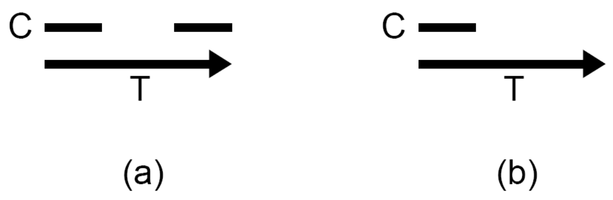


Figure 2.13: Component C participates in transaction T

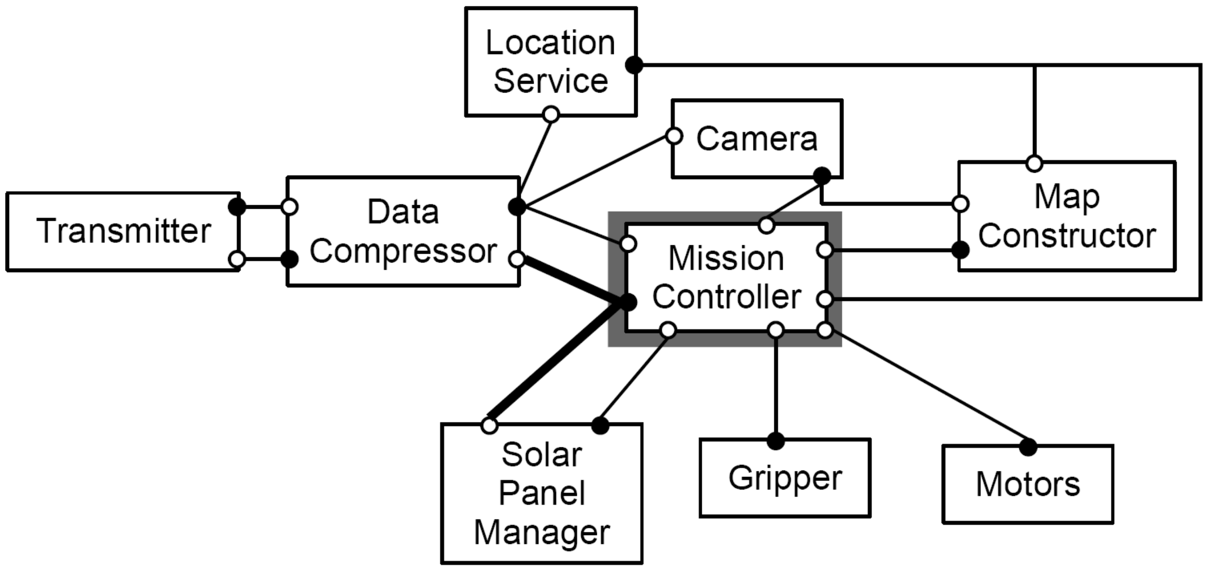


Figure 2.14: Tranquilised components

This weakened condition means that only the component to be replaced need be stopped. In most cases, its neighbours can continue operation. However, there are situations where tranquility will never be reached, such as when two transactions overlap such that the component is perpetually in situation (a). In this case, the neighbours must be made passive[8] as under quiescence. Tranquility is very effective in the rover example (Figure 2.14) since most of the connections to the

[8]"tranquilised"

controller are outbound. It then suffices to wait for the data compressor and solar panel manager to report that they will not reuse the controller. In other words, they must “lock” the controller if they intend to use it multiple times in a transaction.

2.6 Summary

Much of the work described above addresses one or more of our requirements. Section 2.3 covered the most relevant work which addresses the question of what changes an adaptation consists of. Table 2.1 summarises the review, indicating which approaches come closest to satisfying our aims.

Approach	Declarative autonomy	NF properties	Structural properties	Safety	Distribution	Comment
Dynamic Darwin	D2	-	-	-	-	Adaptation restricted
Dynamic Wright	D2	-	Y	-	-	
Dynamic Acme	D3	Y	Y	-	-	NF motivates adaptation
Rainbow	D3	Y	Y	-	-	NF motivates adaptation, almost D4
Aura	D4	N1	-	-	-	
Georgiadis	D3	-	Y	-	Y	
Genie	D3	-	-	-	-	
Critics	D3	-	Y	Y	-	
Policies	D3	-	Y	-	-	
Graph Grammars	D2	-	Y	-	-	
Spec. Retrieval	D4	-	-	-	-	
Service Brokering	D4	N2	-	-	-	
MADAM	D4	N3	Y	-	-	
Arch. Planning	D4	N3	Y	Y	-	NF pertains to deployment

Table 2.1: Summary of related work

Four approaches in particular stand out: Rainbow, the work of Georgiadis, architectural planning and service brokering (Dino). Rainbow uses non-functional information to inform the adaptation process, and respects structural constraints in doing so. However, Rainbow remains largely procedural in its use of repair strategies.

The architectural planning of Arshad *et al.* is fully declarative, satisfying the first of our requirements, and it appears that it would address safety and structural constraints. The use of NF information is limited in that it only relates to the deployment of the configuration rather than its application behaviour. However, the most significant limitation is the performance cost of generalised planning.

The Dino service broker performs matching on functional and non-functional properties, and does not require any procedural specification. Although it does not address the problem of safety, its main drawback is the lack of global knowledge – selection is performed on a component-by-component basis and is not verified against structural constraints.

The work of Georgiadis stands out as the only proposal which explicitly addresses the issues of decentralisation, namely of ensuring that nodes see a consistent view of the global configuration and co-ordinate their changes. The adopted approach however did not overcome the performance limitations of a centralised system.

In Section 2.5 we described work which specifically addresses the requirement for safety during the adaptation process. The original quiescence work entails substantial disruption to the running system, while the subsequent works attempt to minimise this. Wermelinger's approach has the disadvantage that something must be known about the implementations to determine which transactions are related. Tranquility preserves the black box principle by requesting that components identify which of their dependencies they are going to reuse within the current transaction, so that the adaptation system can wait for a point at which the component to be replaced is no longer required. The advantage is that only that component need be stopped, unless the application takes too long to reach the safe point (in which case quiescence is applied).

We can see from this survey that there is a need for solutions which meet D4 by entirely eschewing procedural specification (as Dino and architectural planning do), while simultaneously accounting for NF properties (requirement 2) and structural properties (requirement 3) of the configuration. Very few of the existing works include a safety protocol (requirement 4), and only one specifically addresses the issues surrounding decentralisation (requirement 5). With this in mind, the algorithms and protocols detailed in the subsequent chapters constitute a proposal to meet each of our requirements. The centralised assembly process in Chapter

4 addresses requirements 1 to 3, the extension of tranquility given in Chapter 6 addresses requirement 4, and the decentralised formulation of the assembly process given in Chapter 5 addresses requirement 5.

Chapter 3

The Three-Layer Model

As alluded to in previous chapters, our work is founded upon the three-layer conceptual model of Kramer and Magee [KM07] which was developed from a similar work in the robotics field [Gat98]. This model provides a framework for handling self-adaptive concerns wider than those specific problems addressed in our work, and provides a context in which to demonstrate the applicability of our solutions. Specifically, our approach sits in the middle layer, taking input from the layer above, and driving the layer below[1].

3.1 Conceptual Model

The three-layer model is depicted in Figure 3.1. The primary intent of the model is to stratify the decision-making capability of a system implementing the model so that the most expensive operations are performed the least frequently. This is reflected in the level of abstraction employed, and the amount of deliberative analysis performed, in each layer. This ensures that the system can react quickly to highly variable (possibly continuous) environmental properties, and fall back

[1]The reification of aspects other than the techniques and algorithms of the middle layer were undertaken as joint work.

upon expensive analyses only when necessary. Whereas similar models [KC03] employ a single feedback control loop, incurring the same cost for each adaptation, the three-layer model has (depending on the interpretation) three feedback loops which operate at different rates.

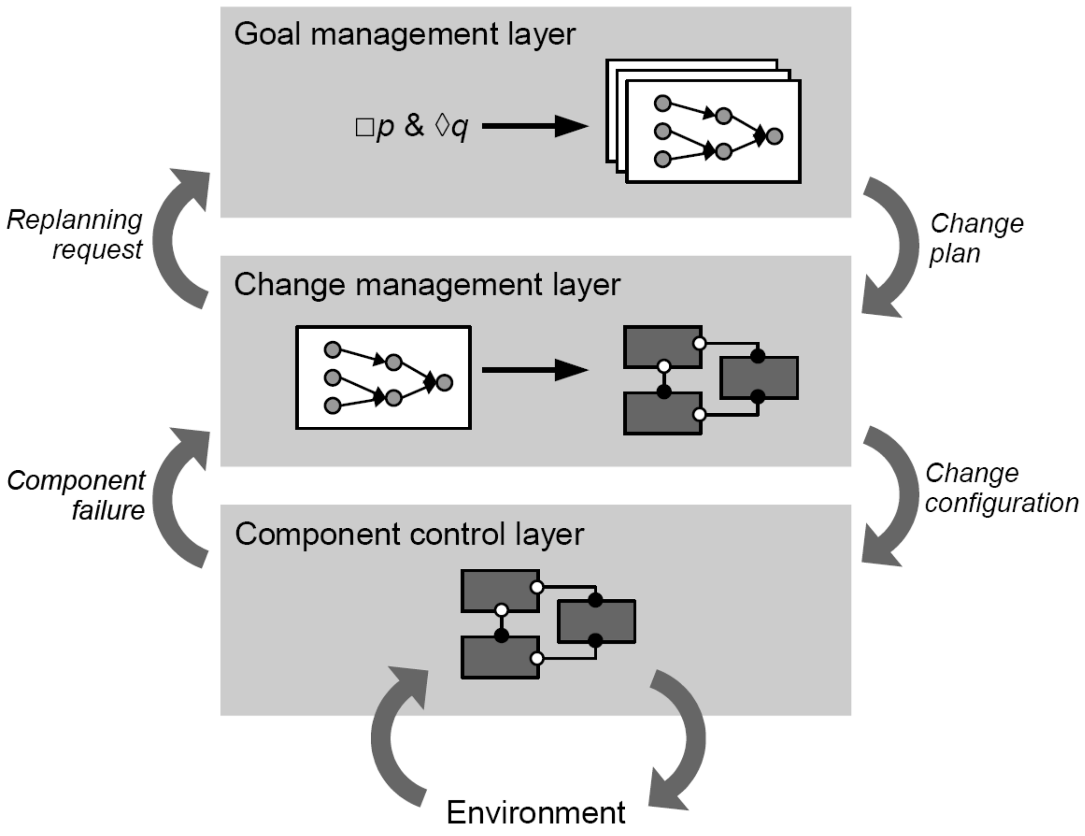


Figure 3.1: The three-layer model for self-adaptive systems

The first of these loops is between the environment and the *component layer* (or control layer). In this layer reside several components which implement the fundaments of the application behaviour and which react quickly to small environmental changes. For instance, a robotics application would handle obstacle avoidance in this layer since a rapid response is imperative.

The second feedback loop lies between the component layer and the *change management layer*. The change management layer has two concerns. The first is to instruct the components to perform high-level actions in response to environmental conditions. The sequence of such actions (in other words, a *plan*) should lead the system to meet its high-level goals, which are handled in the layer above.

The second concern of the middle layer is to manipulate the configuration of components in order to adapt to component failures (which may result from unhandled environmental changes or implementation bugs). It is this problem which is addressed in our work. Both functions of the change management layer are likely to be invoked less frequently than those of the control layer, since reconfigurations are discrete, substantial changes; and plan actions (and the conditions on which they depend) are abstracted from the fine-grained *ad hoc* code which the application components consist of. For example, in the robotics domain again, if a robot is trying to reach a target location using a GPS component, and that component fails (perhaps the GPS signal is lost), then an architectural change could be made to use a different means of getting to the target, such as following a wall. These abstract behaviours would be encapsulated in one or more components which can be replaced without necessitating a change to the current plan.

The final feedback loop lies between the change management layer and the *goal management layer*. The purpose of this layer is to synthesise the overall behaviour of the system in the form of a plan, which is to be executed in the layer below. The plan is derived by analysing an abstract model (called a *domain model*) of the system and its environment, in conjunction with a specification of the goal to be achieved. Feedback comes in the form of replanning requests when a given plan fails to achieve the goal[2]. Such requests are expected to be infrequent, given the numerous opportunities for adaptation in lower layers afforded by the level of abstraction in the plan, avoiding the expense of replanning.

The result of this arrangement is a system which can derive a high-level behaviour from an abstract goal, and assemble (in a top-down manner) a configuration of application components to execute that behaviour, and which provides means for adaptation in each layer to discourage the use of expensive mechanisms to perform trivial changes.

[2]Replanning can also occur if the goal is changed.

3.2 Reification

Having covered the model in abstract terms, we now describe a particular reification of the model which uses a declarative configuration assembly process in the change management layer, directed by the goal management layer. This assembly process, as indicated by the requirements given in the introduction, is the focus of subsequent chapters.

3.2.1 Planning

In the goal management layer, *reactive* planning is used [SHMK08, Sch87]. Whereas a traditional linear plan consists of a sequence of actions, blindly executed in the expectation that the system will move from some starting state to a goal state, a reactive plan observes the state of the world after every step in order to determine what action to take next. A reactive plan, then, consists of a map from states to actions, which imposes no execution order except that imposed by the state changes which occur in the environment. Such a state change may be unexpected, perhaps as a result of some part of the environment failing to match the assumptions made about it, or perhaps as a result of an unmodelled aspect of the environment changing, such as an unwitting human tramping in and breaking something. The unexpected state may be advantageous to the system, by putting it closer to the goal state, or it may be completely unrelated to the actions which the system has performed thus far. It is this latter case which would have required replanning if a linear plan were to be used. Reactive plans deal with the problem by including an action for every state from which the goal is reachable. In other words, whatever state the environment takes, the system knows how to respond, providing it is still possible to achieve the goal.

Domain Model

In order to generate a reactive plan to achieve a goal, the environment and the actions that the system can perform within it must be described precisely and unambiguously. This description is called the *domain model*. The domain model includes a set of propositions P which represent discrete facts which may hold in some states of the environment and not in others. A valuation of these propositions uniquely determines a logical state of the environment. Ordinarily, the truth of propositions will be sensed from the runtime environment to determine what the current (logical) state is. Other propositions may refer to internal system properties. For example, in the robotics domain a proposition such as $atLocation(1)$ might hold when the robot can detect it is at symbolic location 1[3].

In addition to the set of propositions, a set of transitions $Trans$ labelled with actions that the system can perform are defined. These transitions determine how the environment is *expected* to change from one state to another after the system performs an action. Actions are assumed to be instantaneous. For example, a transition may specify that when the state is $atLocation(1) \wedge \neg holdingBall$, and the action $grabBall$ is performed, the environment is expected to move to the state $atLocation(1) \wedge holdingBall$. A given action may be associated with multiple transitions where the result of the action is non-deterministic. These transitions mean that the domain model can be considered to be a labelled transition system[4] (LTS) where states are labelled with a valuation of the set of propositions.

More formally, a domain model is a tuple:

$$Domain = \langle P, States, Actions, Trans \rangle$$

where P is the set of propositions, $States \subseteq 2^P$ is the set of valuations of propositions

[3]The interpretation of propositions and the consequent quantisation of sensed data is a matter for plan execution.

[4]Actually a Kripke structure.

that determine environment states, and $Actions$ is the set of actions used in the transition relation $Trans \subseteq States \times Actions \times States$.

The set of actions can be partitioned into Sys, the actions that the system controls directly; Env_{Dep}, the set of *causally-dependent* actions performed by the environment; and Env_{Ind}, the set of *causally-independent* environment actions. *Causally-dependent* actions are events expected to happen in the environment after a system action has initiated some behaviour. For example, the system may initiate moving towards a target using an action $startMoving$, and completion of the behaviour is notified by the environment performing the action $\_arrivedAtTarget$[5]. *Causally-independent* actions are those which can be performed by the environment at any (applicable) time. For example, in a system which can detect rainfall but has no control over it, $\_startRaining$ may occur at any time when it is not already raining.

The distinction between kinds of actions arises from the need to include system behaviours which are not instantaneous and have duration. Such behaviours have an initiating action of the form $beginBehaviour$, which makes a proposition such as $performingBehaviour$ true. To complete the model, there must be a transition from the state where $performingBehaviour$ is true, and this transition is labelled with a causally-dependent environment action of the form $\_behaviourComplete$ (which sets $performingBehaviour$ to false). For example, the system action $startMoving$ might set the proposition $moving$ to true. After some time the environment action $\_arrivedAtTarget$ happens and sets $moving$ to false. However, this distinction between environment and system actions is not necessary for assembly, and so in the following references to actions should be taken to indicate Sys, unless otherwise specified.

Our implementation [SHMK08] previously made use of NPDDL[6] [BCDLP03] to specify the domain model. In this formalism, a domain model is a set of actions with pre- and post-conditions over the domain propositions, with the meaning that

[5]We use an underscore to denote environment actions.
[6]Non-deterministic planning domain definition language.

in states where the pre-condition holds, it is reasonable to perform the action, and doing so will lead to a state where the post-condition holds. In effect, each of these 'rules' defines several transitions labelled with the given action from all the states which satisfy the pre-condition.

A more expressive means to specify the domain model [HSMK09], however, is as a set of *fluents* (the domain propositions) and a number of constraints written in linear temporal logic (LTL). These constraints can specify pre- and post-conditions but also constraints that hold in all states, such as the fact that a robot can only occupy one location at once (making the relevant fluents mutually exclusive). An example domain model, in the syntax of the Labelled Transition System Analyser (LTSA)[7], is given below:

```
fluent holdingBall = <{grabBall}, {dropBall}> initially false
fluent atLocation1 = <{arriveAtLocation1}, {moveToLocation2}>
fluent atLocation2 = <{arriveAtLocation2}, {moveToLocation1}>
...
constraint GrabPre = ([] (!X grabBall W !holdingBall && atLocation1))
constraint SingleLocation = ([] (!(atLocation1 && atLocation2)))
```

Each fluent declaration gives the list of actions which make the fluent true, and the actions which make the fluent false. In this case, the fluent `holdingBall` is specified to become true when `grabBall` is peformed, and become false when `dropBall` is performed.

The constraint `GrabPre` specifies the pre-condition of the action `grabBall`. Here, the operator `[]` means "for all states", `X` means "in the next state" and `W` is the *weak until* operator, which in this context can be read as "unless". The meaning of `GrabPre`, then, is that for all states, `grabBall` is not performed in the next state unless `holdingBall` is false and `atLocation1` is true. Notice that actions are treated as propositions that hold in the *target* state of transitions labelled with the same action. It is not necessary to specify the post-conditions of actions, since

[7]`http://www.doc.ic.ac.uk/ltsa/` The grammar can be found in the appendix.

these are covered by the fluent declarations.

The second constraint `SingleLocation` uses the LTL syntax to state that there is no state where both `atLocation1` and `atLocation2` hold.

Plan Generation

Once a domain model has been defined and a goal is given, a plan can be generated. A reactive plan is a map from states (as above) to actions: $Plan : 2^P \rightarrow Actions$. $Plan$ contains an entry for every $s \in ReachableStates$ where $ReachableStates$ is the set of states from which the goal is reachable. States $s \notin ReachableStates$ are of no interest to the plan since there is no way (described in the domain model) for the goal to be reached from them. For some $a \in Actions$, $Plan(s) = a$ only if $Trans(s, a, s')$ for some $s' \in States$.

Execution of the plan involves sensing the truth of propositions P to determine the current state and then executing the action associated with that state. The order of actions is thus solely determined by the response of the environment to the actions. Strictly, this sort of plan is called a *strong cyclic* plan since there is the possibility that it will never terminate. This could happen if the environment does not change after an action is performed, leading the system to repeat the same action (or engage in longer cycles of actions).

The first step to generate such a plan is to identify the states in which the goal holds. These are referred to as goal states. Goal states are identified using LTSA by asserting the negation of the goal (given as a further LTL formula), generating the Büchi automaton corresponding to the negation [GM03, MK00] and composing this with the LTS of the domain. LTSA then highlights the states in which the negation of the goal is violated, that is, the states in which the goal is satisfied.

The reactive plan is subsequently generated by an algorithm from the planning-as-model-checking community [GT99]. This algorithm starts with the set of goal states, and adds states which lead to a goal state via a single transition. The transitions leading into the set of goal states are added to the plan. This process repeats with an ever-increasing set of states (and transitions) until there are no more states that can be added. The set of transitions collected at that point constitute the reactive plan.

Figure 3.2 shows two steps of plan generation. In (a), state 1 is added to the plan because its exiting transition leads to the goal state (in black). In (b), states 2 and 3 are added to the plan because their transitions lead to states already selected (state 1).

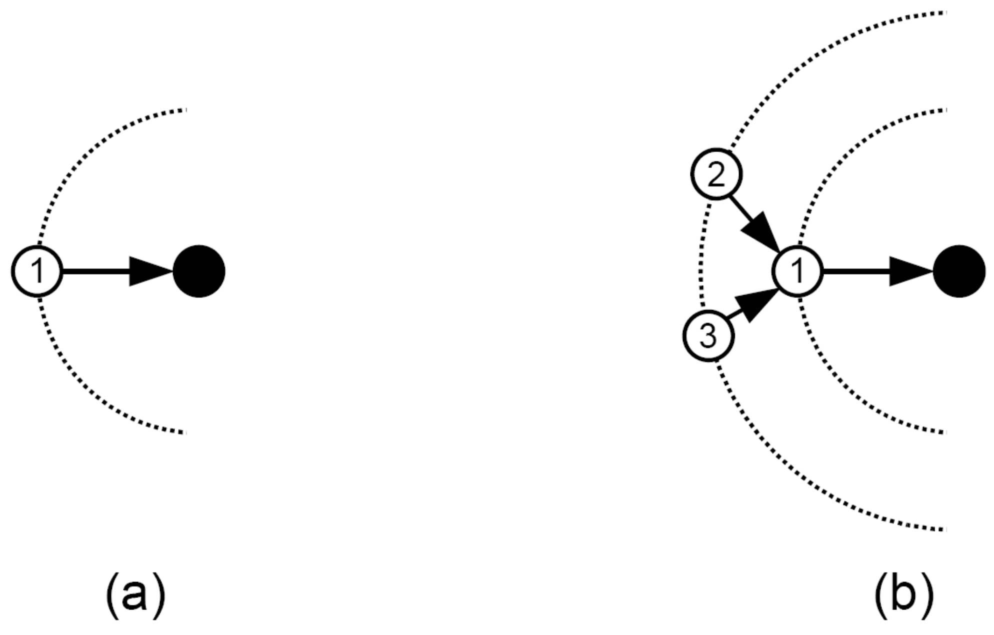


Figure 3.2: Plan generation

Notice that it is possible for a state to have more than one transition that leads into the set of collected states. Since the plan requires just one action associated with every state, the algorithm must choose between them. In this case the algorithm favours the transition which belongs to the shortest path (in terms of the number of transitions). Other strategies can be applied including transitions weighted with NF properties such as cost or reliability. Figure 3.3 shows an example wherein state 3 has two transitions, one to state 2 (indicated with a double arrow-head) and one to state 1. The transition which is selected for the plan is that to state 1, since this path leads to the goal in fewer steps.

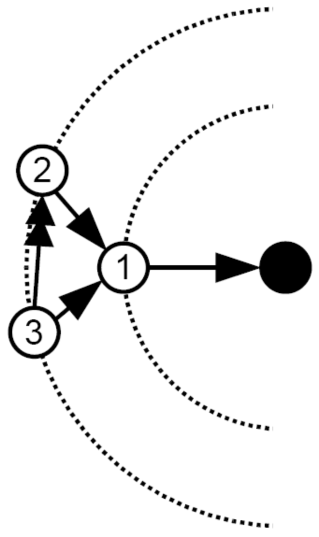


Figure 3.3: Multiple exiting transitions

Plan generation can be regarded as pruning the domain model to remove (i) states from which the goal can never be reached and (ii) longer paths where a non-deterministic choice between actions is present. The result of pruning is a set of trees, each rooted at a goal state.

At this point it is worth noticing that by handling architectural concerns in the layer below goal management, which leaves only the high-level application behaviour to be handled in the plan, the space of propositions and states is reduced, thus simplifying the planning process (cf. Section 2.3.14). In the general case, the number of states grows exponentially with the number of propositions ($|States| = 2^{|P|}$). The state explosion problem [GM03] can limit the scalability of reactive planning, and so restricting the planning domain in this way can provide significant performance benefits. In addition, the abstraction of architectural concerns from the plan allows the system to adapt, by changing components, without having to perform an expensive replanning step.

Figure 3.4 is an excerpt of a reactive plan taken from one of our case studies [HSMK09] (also see Chapter 7). Each mapping from a state to an action can be seen as a condition-action rule. For instance, rule 12 states that when the proposition `MovingEast` is false and `DoorOpen` is true, and so on, the system should perform the action `moveToTarget`. The special action `DONE` indicates that plan execution can terminate, since the goal has been reached.

```
...
// Rule 12
!MovingEast && DoorOpen && !InEast && !MovingToArmEast && !BallAtArmEast
 && !MovingWest && !MovingToTarget && InWest && MovingToArmWest
 && !BallAtArmWest && !HoistingEast && !BallAtTarget && !HoistingWest
-> moveToTarget
// Rule 13
!MovingEast && DoorOpen && !InEast && !MovingToArmEast && !BallAtArmEast
 && !MovingWest && MovingToTarget && InWest && !MovingToArmWest
 && !BallAtArmWest && !HoistingEast && !BallAtTarget && !HoistingWest
-> _arrivedTarget
// Rule 14
!MovingEast && DoorOpen && !InEast && !MovingToArmEast && !BallAtArmEast
 && !MovingWest && !MovingToTarget && InWest && !MovingToArmWest
 && !BallAtArmWest && !HoistingEast && BallAtTarget && !HoistingWest
-> DONE
...
```

Figure 3.4: Example reactive plan

3.2.2 Assembly

Once the plan has been generated, it is presented to the change management layer which firstly uses it to assemble a component configuration, and then executes the plan by observing the state of the environment (and potentially the state of components) and instructing the components to perform the appropriate actions as determined by the plan.

The actions in the plan represent a declaration of the functional requirements[8] which the component configuration must meet: there must be components which can execute each action in the plan. Thus the first step is to use a mapping defined by the user to find a list of interfaces which provide the implementation of actions

[8]We will also call them capabilities.

in the plan. This list of interfaces will then be used as the basis for configuration assembly and adaptation.

The focus of our work is on this assembly process, as described in Chapters 4, 5 and 6. Starting from the mapping of actions, the assembly process analyses the dependencies between components, and uses any structural constraints and non-functional information that the user has provided in order to construct the best component configuration for plan execution.

3.2.3 Components

The component configuration selected by the assembly process is instantiated and operates in the component control layer. Instantiation and binding are handled by the Backbone interpreter [MKM06]. Components are implemented in Java using normal object references as bindings, and are called by the change management layer to execute the actions in the current plan. The details of this implementation can be found in Section 6.7.

3.3 Summary

The three-layer model separates high-level deliberative tasks from low-level behaviours so that adaptations to changes in the environment can be performed at the appropriate frequency and level of abstraction. In the bottom layer, components perform low-level behaviours and react to changes quickly. In the middle layer, behaviours are performed in sequence and composed as a configuration of components. More substantial adaptations can be effected in this layer by changing components. In the top layer, abstract goals are used to generate overall plans. The most wide-ranging and time-consuming adaptations can be achieved by resorting to replanning.

The primary contribution of this work is concentrated in the middle layer and concerns assembling and adapting (through re-assembling) the configuration of components to be used for plan execution.

Chapter 4

Automatic Architectural Assembly

Addressing the functional capabilities required of the system is the first and naturally most important concern of the assembly process. The secondary non-functional concerns guide and constrain the satisfaction of the functional requirements, without changing the fundamental nature of the process. Hence, after stating the relevant assumptions, we first describe the process by which the functional requirements are met, in isolation from the other concerns. Subsequent sections deal with the non-functional concerns.

4.1 Assumptions

Given the wide range of approaches that could be taken in pursuit of our requirements, it is necessary to make certain assumptions to focus our effort on particular solutions.

The first and largest assumption is that the problem of adaptation is to be tackled at the level of software architecture, where adaptations are achieved by the selection of different components and changing the connections between them. As indicated in the introduction, there are several other levels at which adaptation can be considered, including parameter adjustment and dynamic update of classes. An architectural approach allows us to consider a wide range of adaptations from single components to complete changes of functionality, and renders several lower-level considerations unnecessary.

Within the architectural approach, we suppose the existence of an extensible repository of architectural descriptions of components, subsets of which can be instantiated to create a working configuration. The form of these descriptions closely follows Darwin, with unnecessary features omitted. Each component is described by its name and a set of ports. Each port may provide or require a single named interface. In the following, the set of provided interfaces of a component c is denoted by $prov(c)$, and the set of required interfaces by $req(c)$.

The provision of an interface $i \in prov(c)$ indicates that the component provides some functionality associated with the name i. Thus, it is the interfaces that determine which components can be used to satisfy a functional requirement. To this end, we assume the assembly process is furnished with a mapping between the functional requirements (which may be actions from a plan, as above), and the interfaces which can be used to meet those requirements.

Furthermore, for the purposes of resolving dependencies between components, we assume that interface names are sufficient. In other words, we assume the interfaces form a closed ontology for the application domain. We are not concerned with the problem of providing wrappers [YS97] or proxies so that compatible but differently-named interfaces can interact, nor are we concerned with more detailed matching of interfaces using semantic descriptions (as in Section 2.3.11).

Our final structural assumption is that it is reasonable to bind a requirement to *any* matching provision, and that where alternatives exist they are functionally equivalent. This means that connections can be generated trivially and thus omitted from our definition of a configuration. We recognise that these alternative bindings (topologies) may have different non-functional properties, but regard the functional concerns as paramount and the NF concerns as secondary. This also makes it unnecessary to consider multiple instances of each type of component, since a single provision can satisfy many requirements. Where multiple instances are necessary (perhaps to meet a non-functional obligation), this apparent restriction can be overcome by a renaming of components and interfaces so that the different instances can be distinguished.

For the incorporation of non-functional information, we limit the expressiveness of annotations to a constant value referring to a single component. The assembly process does not consider the meaning of each annotation (so a property labelled "memory" does not entail special consideration of the physical memory available) and assumes each property is orthogonal. The local nature of annotations means that functionally-equivalent components can be ranked to make a choice and the lack of semantics frees us from having to introduce specialised procedures for each property, and allows the designer freedom in using whatever properties are relevant to the domain.

4.2 Functional Concerns

In contrast with other approaches (particularly architectural planning [AHW07]), we do not attempt to choose between the manifold combinations of connect, disconnect, create and destroy commands in order to construct or modify a configuration. This unnecessarily increases the complexity of the problem, as is clear from the performance of architectural planning. As indicated above, we leave the bindings between components implicit, leaving the core problem of choosing which components are needed. A configuration is thus a set of components (component types), where create and destroy commands are implied by the difference between two configuration sets.

The first task of the assembly process is to consider the set of functional requirements or capabilities. To find component implementations providing these capabilities, a mapping from capabilities to interfaces is used:

$$Implements : Capabilities \rightarrow P(Interfaces)$$

Given a set of desired capabilities Cap, the complete set of required interfaces can be determined thus:

$$Implements(Cap) = \bigcup_{c \in Cap} Implements(c)$$

Finding implementations of these interfaces is the first part of the assembly process. In the general case, the selected components have requirements which must be satisfied by searching for implementations of the further required interfaces (this is termed the *dependency analysis*). Where two interfaces provide the same functionality, their names must match, or the provision p must be a nominal subtype [Lis87] of the requirement r: $p < r$ (that is, it can provide more functionality but not less).

4.2.1 Dependency Analysis

The problem to be solved by the dependency analysis can be characterised as finding an $arch$ (the set of components in the configuration), given $Implements(Cap)$, subject to two constraints:

$$\forall r \in Implements(Cap) : \exists c \in arch : \exists p \in prov(c) : p \leq r \tag{4.1}$$

$$\forall c \in arch : \exists r \in req(c) \longrightarrow \exists c_2 \in arch : \exists p \in prov(c_2) : p \leq r \tag{4.2}$$

The first constraint states that for all initial requirements, there should be an implementing component, and the second constraint is simply that all component requirements are satisfied. Implicit in the second constraint is the expectation of cyclic dependencies. If component c depends on c_2, and c_2 requires something provided by c, then this latter requirement is satisfied by c, instead of introducing a different component with the same provision (or another instance of c). We call a configuration satisfying these constraints *complete*.

To see how these constraints can be satisfied, consider the following example in which $Implements(Cap) = \{i_1\}$. There are three components c_1, c_2 and c_3. In the descriptions of these components, $prov(c_1) = \{i_1\}$, $req(c_1) = \{i_2\}$; $prov(c_2) = \{i_2\}$, $req(c_2) = \{i_3\}$; and $prov(c_3) = \{i_2, i_3\}$. Figure 4.1 shows this information using the Darwin notation that we use throughout. Then a valid configuration must include c_1 since it is the only provider of i_1, and there must be further components to satisfy the i_2 requirement of c_1. One valid solution is $\{c_1, c_2, c_3\}$, while $\{c_1, c_3\}$ is another.

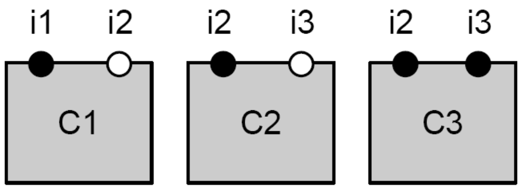


Figure 4.1: Example components

4.2.2 A Constraint Satisfaction Problem

The above problem can be posed to a tool such as Alloy[1], which can then search for a solution to the constraints. In Alloy, the components and interfaces are expressed as *signatures* with particular components represented as singleton extensions of the abstract signatures[2]:

```
abstract sig Interface {}
abstract sig Component {req: set Interface, prov: set Interface}
```

[1]http://alloy.mit.edu
[2]Chatley [CEM03] provided a similar formulation, but with explicit bindings.

```
one sig ExampleInterface1 extends Interface {}
one sig ExampleInterface2 extends Interface {}
one sig ExampleComponent extends Component {}
{
  prov = ExampleInterface1 + ExampleInterface2
  req = ExampleInterface1
}
```

This defines two interfaces, `ExampleInterface1` and `2`, and a component `ExampleComponent` which provides the set {`ExampleInterface1`, `ExampleInterface2`} and requires the set {`ExampleInterface1`}. A configuration is defined as a set of components:

```
sig Configuration in Component {}
```

And finally the two constraints of a valid solution are given (we ignore subtyping for simplicity). The `userRequirement` is stated as a *fact* (since it must hold), indicating `ExampleInterface1` is the only functional requirement (this constraint is a specialisation of 4.1). The *predicate* `validConfig` allows us to ask Alloy to find an example configuration where the constraint 4.2 holds:

```
fact userRequirement { some c:Configuration | ExampleInterface1 in c.prov }
pred validConfig { all c:Configuration, i:c.req |
                   some d:Configuration | i in d.prov }
run validConfig
```

Unfortunately, this proved to be substantially inefficient. Figure 4.2 shows the time complexity was exponential with the number of components. This is to be expected since Alloy translates the problem to one of boolean satisfiability (where a (logical) model is a set of components). The underlying SAT solver attempts to satisfy the constraints by selecting different subsets of components without paying particular attention to the dependency graph. In a sense, Alloy performs an undirected search whereas greater efficiency can be achieved with a directed search over the dependency graph.

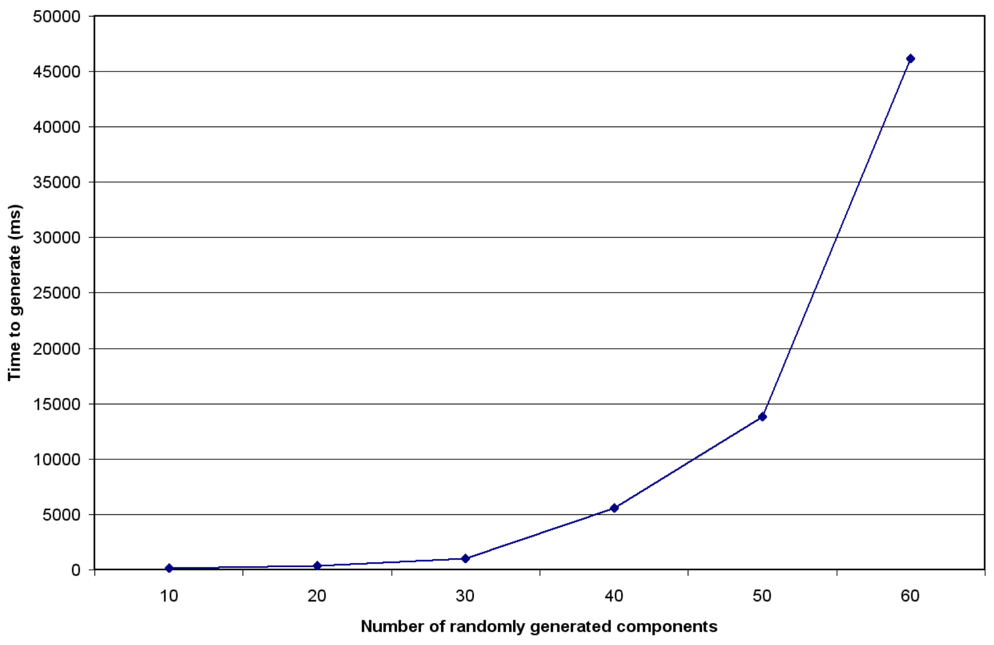


Figure 4.2: Alloy performance

4.2.3 Dependency Analysis Algorithm

We have developed the algorithm shown in Figure 4.3 to overcome the limitations of undirected search as above. The algorithm performs a depth-first search over the dependency graph between provided and required interfaces.

The algorithm is expressed as a function which takes an existing configuration (called `arch`), and a set of names of desired interfaces. The `arch` parameter may be the empty set, to generate an initial configuration. In the first branch, the algorithm checks whether there is a component in the current configuration which already provides the desired interface. If so, the interface can be removed from the set of desired interfaces. The function $prov(x)$ returns a set of interfaces provided by component x, and $req(x)$ does likewise for requirements. In the second branch of the algorithm, the repository of known components (here expressed as a set called $Components$) is searched for components which provide the desired interface, and are available (this small restriction enables the approach to deal with dynamic availability). If there are no providers of the desired interface, the special value `null` is returned, indicating that no configuration can be found. The function is called

```
Construct(arch, interfaces)
  $\forall$ i $\in$ interfaces
    if ( $\exists$ $c$ $\in$ arch : $\exists$ p $\in$ $prov(c)$ : p $\leq$ i)
      interfaces := interfaces - {i}
    else
      providers := {$x_j$ : $x_j \in$ $Components$ $\wedge$
                           $\exists$ p $\in$ $prov(x_j)$ : p $\leq$ i $\wedge$
                           available($x_j$)}
      archs := {Construct(arch $\cup$ $\{x_j\}$, interfaces $\cup$ $req(x_j)$)
                : $x_j \in$ providers}
      if ( $\exists$ a $\in$ archs : a $\neq$ null )
        interfaces := interfaces - {i}
        arch := a
      else
        return null
  return arch
```

Figure 4.3: Configuration generator

recursively to find the requirements of each of these components, returning a set of configuration choices in `archs`. Some of the providers may have unsatisfiable requirements, indicated by `null`. However, if there is a non-null configuration, then this is selected.

An initial configuration can be generated with the call

$$\texttt{Construct}(\emptyset, Implements(Cap))$$

which will locate components which implement the interfaces in $Implements(Cap)$, and then complete the configuration by selecting further components to satisfy the requirements of those already selected. Subsequent configurations can be generated with a call such as

$$\texttt{Construct}(oldConfig, Implements(Cap))$$

Two particular aspects of this algorithm are deserving of further discussion. The first, which was alluded to previously, is that there may be multiple candidate configurations which satisfy constraints 4.1 and 4.2 in the case when a given interface is implemented by multiple providers. Indeed, it is our assumption that this is the normal case, since this is what gives the system opportunities for adaptation by switching providers. However, given only the functional requirements of the problem, there is little reasonable basis on which to choose between solutions (except perhaps minimality). Thus, the algorithm as given above makes an arbitrary choice by returning the first candidate to satisfy the constraints. This can be seen in the code in branches which lead to the set `interfaces` being empty, causing it to exit the loop and `return arch`. Non-functional preferences provide a better basis for choosing between the candidates, and this is discussed in subsequent sections.

Another interesting feature of the algorithm is that it permits cyclic dependencies between components. Consider executing the body of the loop for a particular `i` which has already been provided by an "ancestor" component (C1 in Figure 4.4). The first branch of the `if` statement checks for this ancestor, and removes `i` from `interfaces`, before going into the branch which searches for a new provider and recursing with its requirements. This means that all requirements for a given interface can be satisfied by a single provision, even where this creates dependency loops.

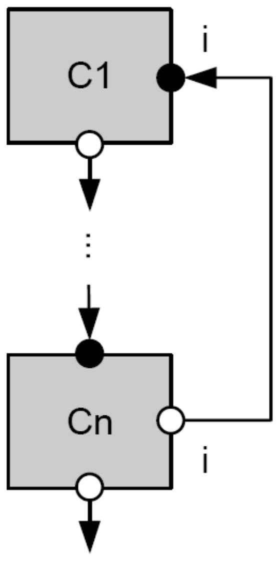


Figure 4.4: Cyclic dependency

In effect, the algorithm computes the transitive closure of the dependency graph for each component directly needed to satisfy a functional requirement. The dependency relation is defined as

$$D(a, b) = r \in req(a) \wedge p \in prov(b) \wedge p \leq r$$

and the notation D^T indicates the transitive closure of D. If $DR^T(a)$ indicates the dependency graph related to component a, that is

$$DR^T(a) = \{(a, y) : D^T(a, y)\}$$

then the configuration generator computes (in the absence of alternatives)

$$\mathit{Config}(a) = \{a\} \cup \mathit{range}(DR^T(a))$$

when a is a component satisfying a functional requirement. Figure 4.5 shows an example dependency graph (without any alternatives) and the selected configuration for the functional requirement `i0`.

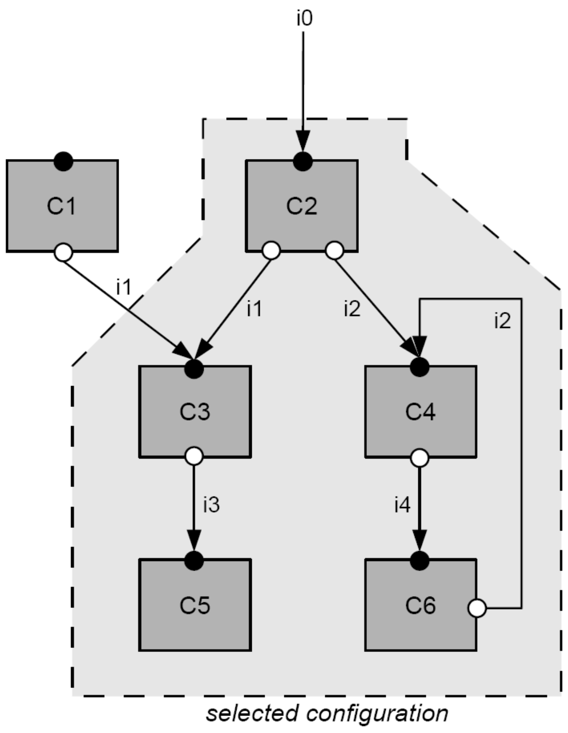


Figure 4.5: Dependency graph and selected configuration for requirement `i0`

Here, $\mathit{Config}(c_2) = \{c_2\} \cup \mathit{range}(DR^T(c_2))$, where

$$D = \{(c_1, c_3), (c_2, c_3), (c_2, c_4), (c_3, c_5), (c_4, c_6), (c_6, c_4)\}$$

and

$$D^T = D \cup \{(c_1, c_5), (c_2, c_5), (c_2, c_6)\}$$

$$DR^T(c_2) = \{(c_2, c_3), (c_2, c_4), (c_2, c_5), (c_2, c_6)\}$$

giving

$$\mathit{Config}(c_2) = \{c_2, c_3, c_4, c_5, c_6\}$$

Once the set of components has been selected, there remains the issue of creating connectors between the components to form a working configuration. Recall that all requirements for an interface can be satisfied by a single provision. Then it suffices to find such a provision in the selected set for each component's requirements and connect them.

4.2.4 Example

Now consider a larger example in which a mobile robot must search for a coloured ball. There are many components available to the robot, providing wide-ranging functionality, including: Koala (which provides interfaces to the motors and sensors), VectorMotionController (which combines different sources of movement), ObstacleAvoider (which enables the robot to avoid obstacles), Webcam, GoToTask (which directs the robot to a target location), BallSurveyor (which uses the camera to detect the ball), and two search patterns, ZigZagSearchPattern and CircularSearchPattern.

The functional requirements are implemented by the interfaces {Surveyor, CollisionAvoider}. In this case, the interfaces are provided by only one component each, giving the set {BallSurveyor, ObstacleAvoider}. The configuration can be completed by calling `Construct` with the requirements of those components, namely {SearchPattern, Camera, MotionController, Sensors}. The Camera requirement is satisfied by selecting the Webcam; the MotionController requirement is satisfied by selecting the VectorMotionController; and the Sensors requirement is satisfied by selecting the Koala. In order to satisfy the SearchPattern requirement, the algorithm must choose between several implementations including ZigZagSearch-Pattern and CircularSearchPattern. Although these differ in their non-functional properties (reliability, coverage), they have the same functional behaviour which is that they perform a search, and so one is selected arbitrarily.

At this stage, the algorithm has selected {BallSurveyor, ObstacleAvoider, ZigZagSearchPattern, VectorMotionController, Koala} but there remain the requirements of the newly-added components to consider. The ZigZagSearchPattern requires a MotionController, but this has already been dealt with. Likewise the VectorMotion-Controller requires a Motors interface which is supplied by the Koala which has already been selected.

The resulting configuration, selected purely on the basis of functional requirements, is shown in Figure 4.6.

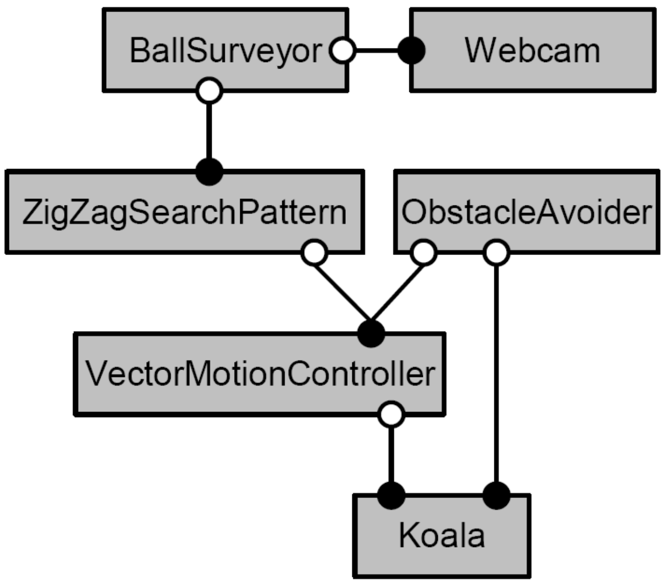


Figure 4.6: Selected configuration for ball survey

4.2.5 Termination

The algorithm can be shown to terminate if one considers the current (global) set of required interfaces, called IFS, which grows as the requirements of new components are added, but shrinks as those requirements are found to be satisfied by the selected components. $IFS \subseteq \mathit{Interfaces}$, the set of all interfaces in the repository. Let $|IFS|$ be the size of IFS. Then in the first branch of the algorithm, $|IFS|$ is reduced by 1 as a requirement has been satisfied.

In the second branch, for each component $x_j \in \mathit{providers}$, $|IFS_j|$ increases by (at most) the number of interfaces in $\mathit{req}(x_j)$ which are not already in IFS. One of these $|IFS_j|$ is selected as the final value of $|IFS|$. Since there are a finite number of possible interfaces ($|\mathit{Interfaces}|$), $|IFS|$ can only increase to the maximum of $|\mathit{Interfaces}|$, since when this is the case, all the interfaces in $\mathit{req}(x_j)$ will already be present in IFS. $|IFS|$ must then decrease by the first branch, causing termination when IFS is empty. Alternatively the algorithm will terminate when it fails to find a component to satisfy a requirement.

4.3 Structural Constraints

In order to incorporate structural constraints into the assembly and re-assembly process the above dependency analysis is extended with a check that the generated candidates meet the system designer's constraints. This is done by invoking an external constraint checker to determine whether or not the candidate is valid. In the present case, this is done using Prolog, which affords the user the full expressiveness of the Prolog syntax to write constraints ([WSKW06] uses Alloy for a similar purpose). Indeed, such a general mechanism does not limit the user to strictly "structural" constraints. Any sort of constraint may be written with the caveat that the only information available from the dependency analysis is the set of component names selected for the candidate configuration, and the dependencies between them. Figure 4.7 shows an example constraint, which describes a ring architectural style, written in the Prolog syntax.

```
allowed(Arch) :- ring(Arch).
ring(Arch) :- \+ (member(C, Arch),
                  \+ ( findall(I, provider(C, I), Provs),
                       findall(J, requires(C, J), Reqs),
                       length(Provs, Ps), length(Reqs, Rs),
                       Ps == 1, Rs == 1 )
                  ),
              member(C, Arch),
              reachable(Arch, C, Cs),
              permutation(Cs, Arch).
% further clauses omitted
```

Figure 4.7: Ring architectural constraint

A candidate `Arch` is checked by evaluating `allowed(Arch)`. When a candidate is vetoed by the constraint check, there are two possible responses: firstly, the candidate can be completely discarded, and the other candidates which arise by virtue of alternative implementations can be considered; secondly, the candidate can be extended in an arbitrary way in an attempt to meet the constraints before falling back to the other normally generated candidates.

Whereas the first option may seem the more intuitive, we have implemented the second, since there are some constraints which can never be satisfied by considering alternative implementations alone (even when it is possible to satisfy the constraints given the set of components available). The simplest example of such a constraint is one which requires the presence of an arbitrary component (for whatever domain-specific reason) which is not required by any other component in the dependency graph. The following constraint is one such example which requires the ObstacleAvoider (which is not required by any other component):

```
allowed(Arch) :- member('koala.motion.ObstacleAvoider', Arch).
```

All candidates submitted to Prolog would be vetoed since the dependency analysis does not select components which are not required by any other. Thus, the only way this constraint could be satisfied would be by adding further components. Unfortunately, since there is no way of knowing (with such an expressive constraint language) what should be added to satisfy the constraints, the system must resort to a blind search through the possibilities. It is worth noting that it is not reasonable to remove components from a candidate in an attempt to satisfy structural constraints since this would make the candidate incomplete (because all components are transitively required).

It might be thought that falling back upon an undirected search returns us to a situation with no performance advantage over general constraint solvers, and indeed this is true in the worst case. However, as shown below, in many cases the efficient dependency analysis gives a performance improvement by taking advantage of truly “structural” constraints.

4.3.1 Performance Compared To Constraint Solving

The approach taken to find valid, complete candidate configurations is thus (i) to perform the dependency analysis (generate a candidate), (ii) to check it against the structural constraints (test the candidate), and if it is rejected, (iii) to add new components iteratively and (iv) to check each extended candidate against the constraints, and only when those options are exhausted, (v) to generate the next candidate from alternative interface implementations. The first complete candidate to pass the constraint check is returned as the chosen solution.

When a new component is added in step (iii), it is returned to the dependency analysis for completion (before the constraints are checked again), in recognition of the fact that the new component will have its own requirements which are best satisfied using the dependency analysis. It is thus the balance between the time spent performing the directed search provided by the dependency analysis and the time spent falling back to undirected search that determines the performance of this approach with respect to general constraint solving. We have found that this balance is controlled by the specific constraints used, and that although our approach has exponential time complexity in the worst case, it is those constraints that might be regarded as truly "structural" that result in a performance improvement.

To illustrate this point, consider the following two examples (which use a simplified syntax). In the first, the component C is part of the required capabilities of the configuration. C depends on D ($c \longrightarrow d$), and another component E depends on F ($e \longrightarrow f$). The structural constraints state that E and F must be included:

$$\begin{aligned} Components &= \{c, d, e, f\} \\ c &\longrightarrow d \\ e &\longrightarrow f \\ \texttt{allowed :- e, f.} & \end{aligned}$$

The dependency analysis will initially produce candidate {C, D}, which fails the constraint check. It then has the option to add E or F. Since E requires F, the new candidates are {C, D, E, F} and {C, D, F}. The first of these will succeed. Here, the constraint is related to the dependency graph and so a solution is produced quickly.

However, in the second example, where the user again requires something provided by C, the constraint is totally unrelated to dependencies:

$$Components = \{b, c, d, e, f\}$$

$$b \longrightarrow c$$

$$d \longrightarrow e$$

```
allowed :- c, f, e.
```

The initial solution is {C}. Since this does not satisfy the constraints, components are added to generate new configurations {C, B}, {C, D, E}, {C, E}, {C, F}. None of these satisfy the constraint, so further components are added to the candidates {C, B, D, E}, {C, B, E}, {C, B, F}; {C, D, E, B}, {C, D, E, F}; {C, E, B}, {C, E, D}, {C, E, F}; {C, F, B}, {C, F, D, E}, {C, F, E}. Three of these now satisfy the constraints, however the first one {C, D, E, F} will be selected.

Figure 4.8 shows two runs of our assembly algorithm (labelled "OriginalRetry") compared with an algorithm which blindly generates subsets of components to test against the constraints ("Subset"), which always shows exponential behaviour, representing the worst case.

In the first case (top), the structural constraint used requires that the candidate includes all components which provided an interface with less than three alternatives (we omit the Prolog representation). Since such components are much more likely to be chosen by the dependency analysis (irrespective of the constraint), our algorithm vastly outperforms "Subset".

The second graph shows the worst case behaviour. Here, the constraint is simply that 50% of the components in the repository are present in the candidate, which gives our approach no opportunities for exploiting the dependency graph.

We expect that in practice the structural constraints used will not be so degenerate as in the second graph, and that they will afford some opportunities for performance improvement over general constraint solving to avoid the exponential

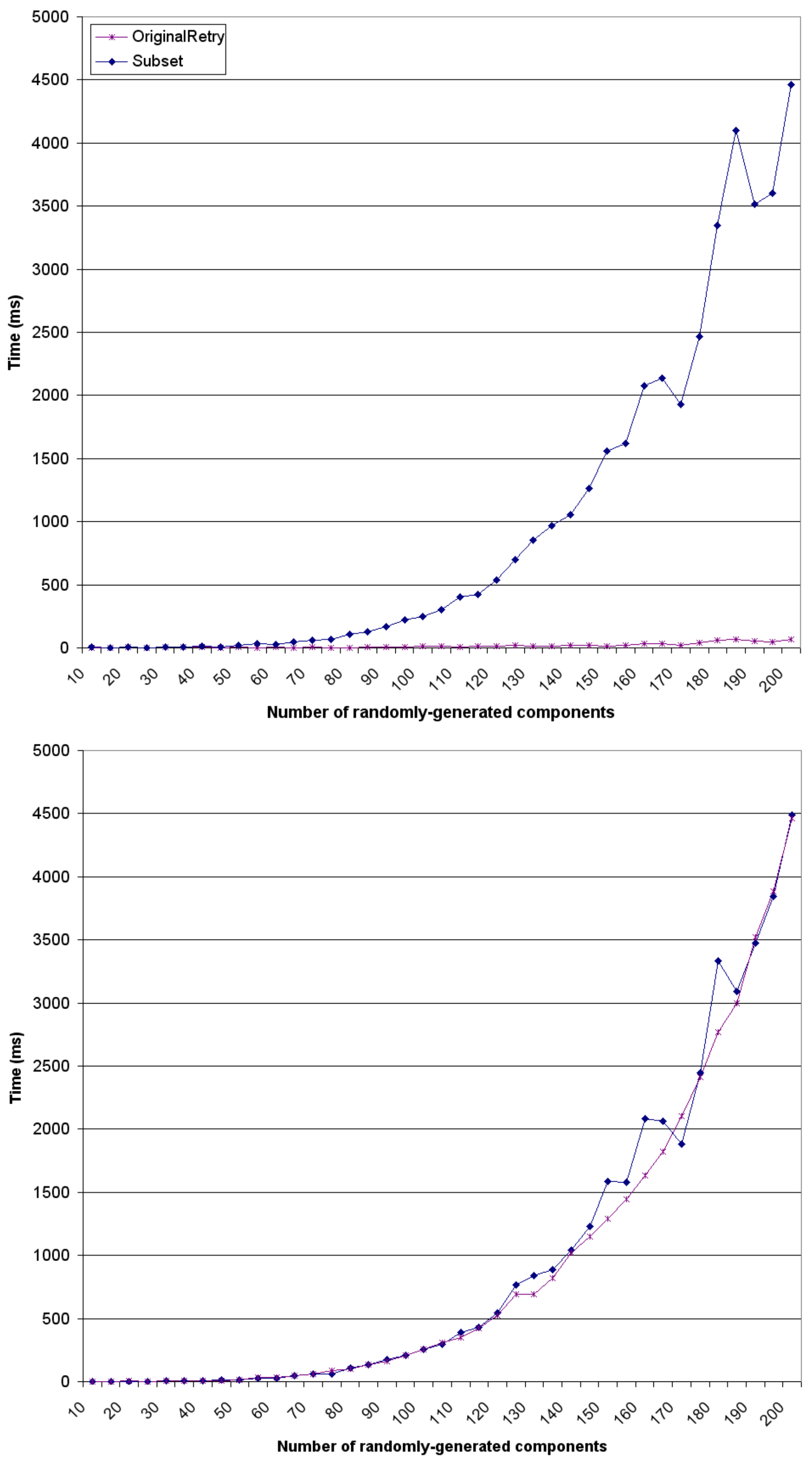


Figure 4.8: Prolog performance

behaviour. Put in this light, our approach can be regarded as a domain-specific heuristic. There is perhaps scope for further investigation into the relationship between the expressiveness of the constraint language (with respect to structural concerns) and the performance of our heuristic.

4.4 Non-Functional Properties

To incorporate non-functional information into the assembly process, the system designer can define a set of non-functional properties, $NFProp$. It is important to note that this set may be entirely domain-specific, since the particular meaning of each property is not necessary during assembly (though it is for monitoring). Table 4.1 gives some examples of the wide range of non-functional properties supported by our generic approach. The user can provide as little or as much information as is available, with the caveat that assembly falls back to arbitrary choice where no information is provided.

This information can be captured using annotations in the ADL description of components in the repository. Two components C1 and C2 could be annotated with `cpu` and `memory` properties as below (the precise syntax is not significant[3]):

```
component C1 [cpu=high, memory=5k]
{
  prov a : I
}

component C2 [cpu=low, memory=100M]
{
  prov b : I
}
```

Here each component is annotated with a set of pairs naming an NF property, and the value of the property for that component. The intended meaning may be

[3]Its grammar can be found in the appendix.

Performance (time complexity, latency, throughput)	The performance of a component is important in many domains, particularly embedded systems and high-load systems such as web servers.
Memory (main memory, permanent storage)	Memory is a particular concern in embedded systems.
Power	Some components have a direct effect on power use (by enabling a specific piece of hardware like GPS), and so in a power-constrained environment like a mobile device this is a critical property.
Bandwidth	This is increasingly relevant as streaming media over the internet (and thus over mobile networks) is becoming more popular.
Reliability	This is a critical property for systems which demand high availability such as banking or military infrastructure.
Security	Again, this is critical in some domains such as military or medical systems.
Monetary cost	Certain components may incur a cost-per-use, such as those which require data over a mobile network.
Fidelity (graphical quality, video compression)	This is an important property for components which deal with the end-user experience, such as rendering engines in games and compression when streaming media.
Usability	Again this is relevant for components pertaining to the user experience. An example might be different input methods on a mobile device where usability needs to be balanced against computational cost.

Table 4.1: Table of non-functional properties

that C1 uses a lot of CPU resources, but little memory, with the contrary holding for C2. We do not ascribe meaning to these natural-language annotations during assembly, and merely rely on the user providing a utility function for each property to produce a value representing how useful that property value is. Meaning is however necessary for monitoring when, for example, it should be possible to update the annotation of C1 from `memory=5k` to `memory=1M` on the basis of data gathered at runtime.

4.4.1 Utility Functions

For each property p in $NFProp$, the user defines a utility function u_p [Bat00] which maps a property value (which may be continuous or discrete) onto a real number between 0 and 1, where 1 represents the most useful value. For example, the utility function for `memory` usage may be defined as

$$u_{mem}(5\text{k}) = 0.9$$

$$u_{mem}(100\text{M}) = 0.1$$

with intermediate values linearly interpolated between these two. Other interpolation functions are possible, such as a sigmoid.

In addition, a relative weight between 0 and 1 is associated with each property which indicates its importance in the domain. This way, the user can indicate their preference for candidates which offer maximal utility in particular dimensions, such as maximal performance or minimal cost. Note that the set of all weights must sum to 1.

With this information, the total utility of a component can be calculated from the property values by taking a weighted sum, resulting in a value between 0 and 1, placing functionally equivalent components in a partial order. In other words, the utility $U(c)$ of a component c is

$$U(c) = \sum_{p \in NFProp} w_p \times u_p(c.p) \tag{4.3}$$

where $c.p$ indicates the value of p for component c (if it is not provided then the utility is assumed to be 1), and w_p is the weight for property p.

Hence, with the additional definition of u_{cpu} as

$$u_{cpu}(low) = 0.9$$

$$u_{cpu}(high) = 0.1$$

and two weights representing the user's preference between the `cpu` and `memory` properties

$$w_{cpu} = 0.6$$

$$w_{mem} = 0.4$$

then the utility of components C1 and C2 can be calculated as

$$U(c_1) = 0.1 \times w_{cpu} + 0.9 \times w_{mem} = 0.42$$

$$U(c_2) = 0.9 \times w_{cpu} + 0.1 \times w_{mem} = 0.58$$

Assuming that `cpu` and `memory` are the only properties of interest, C2 should be chosen over C1 to provide interface I. In general, the components with the highest utility should be selected, but as described in Section 4.4.3, making this decision with only local information may lead to a configuration that is globally sub-optimal.

Having defined the utility of a single component, it is now possible to find the aggregate utility of a configuration in order to choose between candidates by taking the average utility of all the components in the configuration:

$$U_{agg}(arch) = \frac{\sum_{c \in arch} U(c)}{|arch|} \tag{4.4}$$

This method for calculating the utility of a configuration masks a significant assumption which is that the component annotations are correct whether the component works in isolation or in a large configuration; in other words, that NF properties are compositional. It is trivial to conceive of a situation where this is not the case, such as a configuration which involves large numbers of components which claim to be fast, which will no doubt exhibit poor performance. More obscure situations can arise if components compete for resources such as a hard disk.

We aim to mitigate (but not solve) this problem by using monitoring to update the NF annotations, reflecting how the components behave at runtime. A more general solution for accurate, compositional annotations will require significant further work. And yet, even with such assumptions, the task of finding a globally optimal configuration given NF information remains a difficult one.

We now present two schemes for combining these utility functions with the assembly process. The first uses the aggregate utility to choose between a list of complete candidates. The second is a greedy algorithm which uses the utility of components to make local choices. In order to compare these approaches, we consider (i) whether the approach chooses the configuration with maximal aggregate utility and (ii) the overhead in terms of processing time to use the approach.

4.4.2 Aggregate Selection

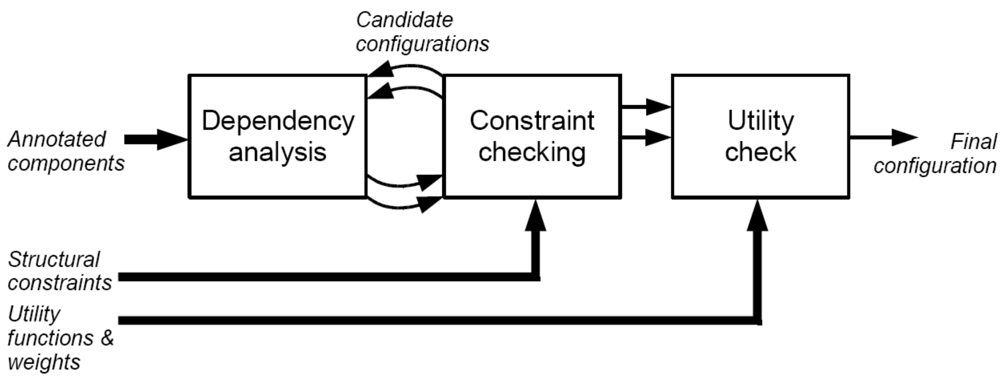


Figure 4.9: Overview of aggregate selection

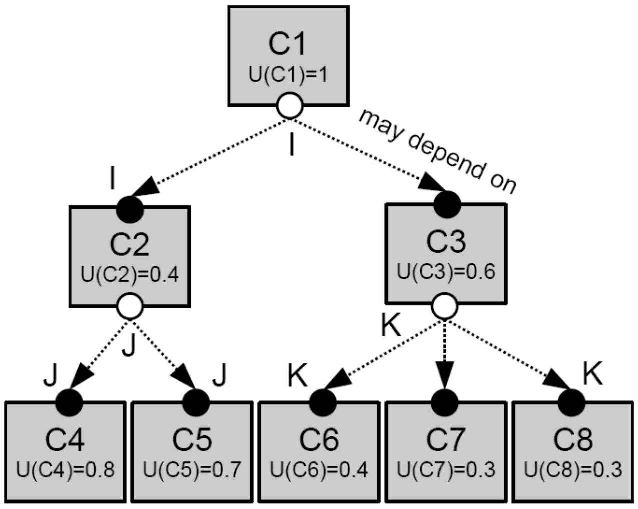


Figure 4.10: Example dependency graph with utilities

Aggregate selection adds a third independent step to the assembly process, which is to compute the aggregate utility (as defined above) for *all* the complete, constraint-satisfying candidate configurations produced by the previous two steps. The candidate with the maximum utility is selected. If there are two candidates with the same utility, the smaller one (in the number of components) is selected. Figure 4.9 shows an overview of the procedure.

For example, consider the components C1 to C8 below. The plan requires interface A, necessitating the selection of C1. C1 then requires C2 or C3 via interface I, which in turn require C4 to C8 via interfaces J and K. This information is depicted in Figure 4.10, which also includes the utility of each component (for simplicity we omit the particular NF properties).

```
component C1 { req i:I; prov a:A }
component C2 { req j:J; prov i:I }
component C3 { req k:K; prov i:I }
component C4 { prov j:J }
component C5 { prov j:J }
component C6 { prov k:K }
component C7 { prov k:K }
component C8 { prov k:K }
```

In this example, there is also an unfavourable (in performance) structural constraint that states that valid configurations do not contain C8. Hence, with aggregate selection, there are 4 candidate configurations which are complete (no requirements unsatisfied) and valid (constraints are satisfied). These are, with their aggregate utilities:

Candidate	U_{agg}
{C1, C2, C4}	0.6
{C1, C2, C5}	0.55
{C1, C3, C6}	0.5
{C1, C3, C7}	0.45

However, to generate this list, it is necessary to generate the full list of complete candidates, and then to discard those which are not valid, since the structural constraints can only be checked for complete candidates.

This means that the configuration {C1, C3, C8} will have been generated and seen to fail the constraint check. Then, since the system is entirely ignorant of what needs to be added or removed to satisfy the constraint, the candidate is extended with further components (C8 cannot simply be removed since this would cause the candidate to be incomplete). In this case, 5 components (C2, C4, C5, C6 and C7) can be added to create a new candidate, giving a maximum of $2^5 - 1 = 31$ extended candidates. Of these, 4 are not considered since they are not complete (that is, those that contain C2 without any of its dependencies), leaving 27 extended candidates such as {C1, C3, C8, C4, C6}, none of which pass the constraint check.

Thus, while this method of selection has the advantage that it produces the solution with the maximum aggregate utility, it necessitates the generation of the full list of candidates, which can sometimes be expensive, depending (as previously described) on the structural constraints employed.

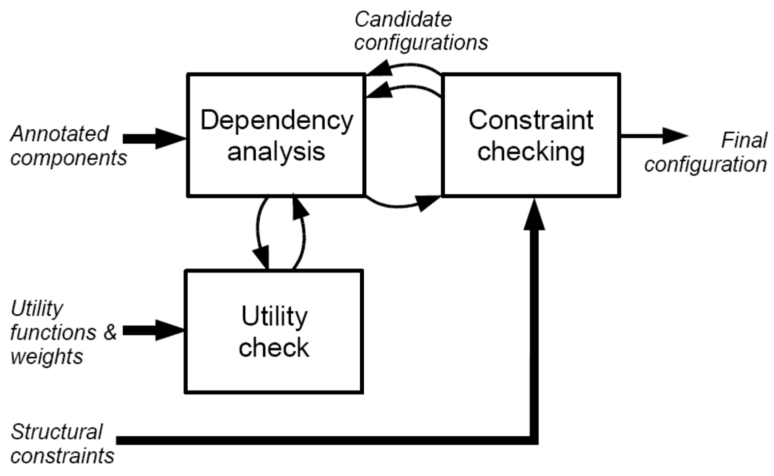


Figure 4.11: Overview of incremental selection

4.4.3 Incremental Selection

Incremental selection integrates the consideration of utility into the dependency analysis, as shown in Figure 4.11. At each step of the dependency analysis, one required interface is under consideration, and all the providers of that interface are found. With incremental selection, the provider with the highest utility is selected (greedily), and the other alternatives are discarded (unless the constraint check vetoes the selection). This is the critical distinction between incremental and aggregate selection: incremental selection is able to stop at the first valid candidate, since utility has guided selection at every step, while aggregate selection must find the (potentially large) list of valid candidates before calculating their utilities.

Taking again the components from the previous section, the dependency analysis starts with the incomplete configuration {C1} and finds C2 and C3 as the implementations of interface I. Since $U(\mathrm{C3}) > U(\mathrm{C2})$, C3 is added to the configuration to give {C1, C3}. Now the implementations of interface K are found, and since the maximum utility among these is $U(\mathrm{C6}) = 0.4$, the next configuration is {C1, C3, C6}. Since this configuration is complete, and passes the constraint check, it is returned as the final configuration. However, notice that its aggregate utility is 0.5, which is less than the maximum. This is because the local choice

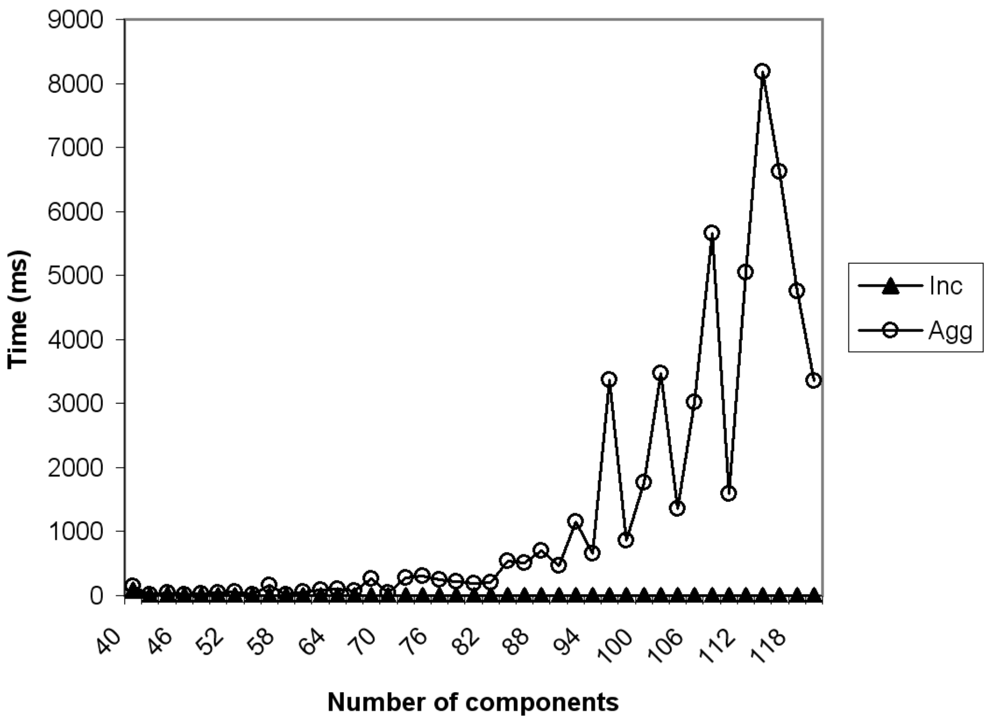


Figure 4.12: Execution times for incremental and aggregate selection using randomly-generated components

between C2 and C3 does not take account of the low utility of the components implicated by choosing C3. In other words, a good choice now may be a bad choice later.

Thus, while this approach is much less costly, its solutions do not maximise the aggregate utility in many cases. In the next section, we quantify the differences between the two approaches by comparing their solutions and performance on a large number of components.

4.4.4 Performance Of Aggregate Versus Incremental Selection

Figure 4.12 shows the time taken (in milliseconds) by each approach to generate a solution for repositories containing between 40 and 120 components. In almost every case, incremental selection ("Inc") takes less than 1ms to produce a solution. This is expected because its behaviour is not dependent on the number of

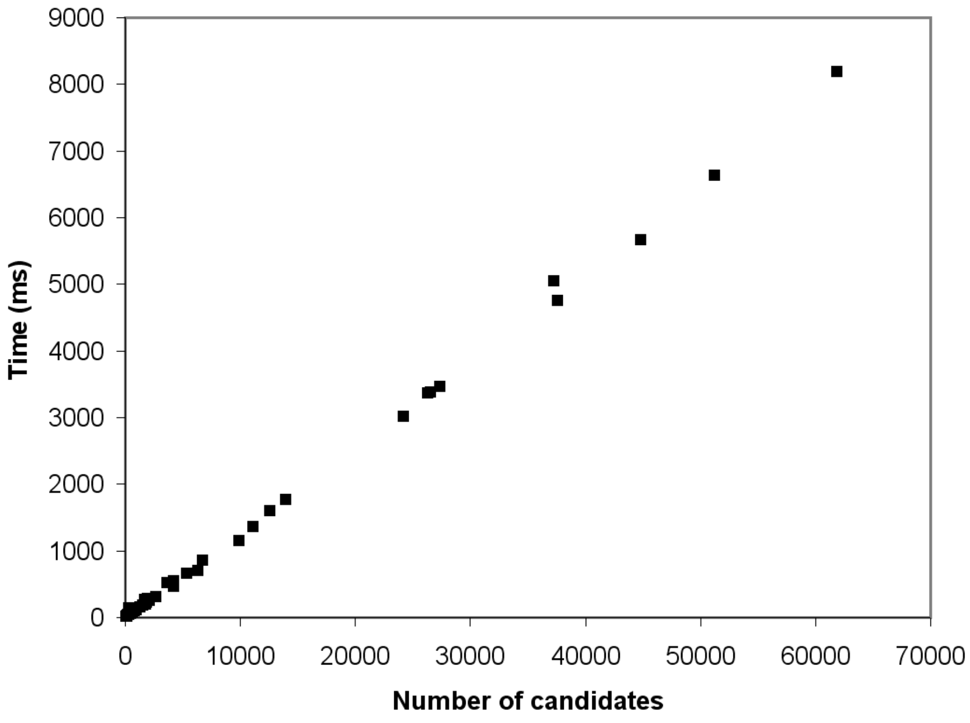


Figure 4.13: Execution times for aggregate selection against the number of valid candidates

components, but rather on the size of candidates, which in these tests is normally less than 10 components.

Aggregate selection, on the other hand, takes increasing lengths of time as the set of components grows. The results for aggregate selection do not form a smooth curve due to the varying number of candidates permitted by each set of components (which is controlled by the number of interfaces and the number of implementations of each interface, in addition to the size of the set). In fact, the time taken by aggregate selection increases linearly with the number of candidates (which increases combinatorially), as shown in Figure 4.13.

In addition to comparing the execution time, we compared the optimality of the chosen candidates with respect to the maximum aggregate utility. Since it is known that aggregate selection chooses the maximum aggregate utility, we can compare the candidate chosen by incremental selection using:

$$optimality(arch_{inc}) = \frac{U_{agg}(arch_{inc})}{U_{agg}(arch_{agg})}$$

where $arch_{inc}$ is the candidate produced by incremental selection and $arch_{agg}$ is that produced by aggregate selection. This allows one to say, given an aggregate utility of 0.4 produced with incremental selection, and a utility of 0.9 produced with aggregate selection, that incremental selection has achieved 44% of the maximum aggregate utility.

Surprisingly, in all the random tests, incremental selection gave solutions between 90 and 100% of the maximum. The average over all runs in Figure 4.12 was 98.9%. This suggests that the computational cost incurred by aggregate selection outweighs the benefit of optimality since runtime adaptation necessitates prompt responses. However, it remains to be seen whether this result holds in a realistic corpus of components.

In any case, the choice between the strategies need not be final, and further strategies may be developed. In fact, the assembly process could switch between them depending on the size of the component repository (using aggregate selection for small repositories only), or could initially use aggregate selection and fall back to incremental selection after a time-out.

4.5 Hierarchy

The remaining aspect to discuss is how the assembly procedure handles a hierarchy of composite components. A composite component can be designed by the programmer to encode domain-specific expertise about which combinations of leaf components are desirable solutions, in other words, to perform some of the assembly work *statically*. Thus, composite components can implicitly encode structural constraints, and can improve the performance of assembly. On the other hand, excessive use of composites may restrict the search space, reducing opportunities for adaptation.

The component repository may contain a mixture of composite and leaf components with identical treatment of dependency and NF annotations. However, when the dependency analysis chooses a composite, it recursively adds all the leaf components specified in the composite to the configuration, effectively flattening the hierarchy. This flattening means that the resulting configuration can be treated in the same manner as a non-hierarchical one for all subsequent steps. The advantage of this approach is that a failed component at any depth in the hierarchy can be replaced while preserving the rest of the hierarchy[4], meaning that the configuration can adapt beyond the user's initial design.

4.6 Summary

In summary, this chapter has described a centralised assembly process which accounts for the user's functional requirements, structural constraints and non-functional preferences to address requirements 1 (declarative autonomy), 2 (NF properties) and 3 (structural constraints), given in the introduction. Functional requirements are addressed using a dependency analysis, which works in conjunction with explicit constraint checks to ensure that only valid candidates are considered. In the worst case, finding a valid configuration can take exponential time relative to the number of available components, but in many cases the directed approach (which considers dependencies first) can vastly outperform generalised constraint solving. NF information is incorporated by using either incremental or aggregate selection. Incremental selection trades optimality in the utility of the result for improved performance. On the other hand, aggregate selection ensures

[4]Although the relevant composites must be marked as failed.

optimality but this comes at some cost in performance. The next chapter describes how this centralised process can be adjusted to work in a decentralised context to achieve greater robustness (in the face of node failure).

Chapter 5

Distributed Assembly

ASSEMBLY as described thus far is a centralised algorithm which requires a complete view of the system in order to derive solutions. This view comprises the list of available components and the complete dependency graph, all the structural constraints and the NF annotations of every component. Although the components may be physically distributed after a configuration has been derived, this part of the problem is invisible to the assembly process.

The two major limitations of a centralised scheme are the lack of reliability and inefficiencies which limit its ability to cope with very large distributed systems. Reliability of a centralised system is dependent upon on the reliability of the central node (in our case the assembly process). Failure of this node can rarely be recovered from. The central node also creates a bottleneck since the computation is limited by the performance of that node and since extensive communication must take place to ensure the central node has a global view of the system.

There is some irony, then, in the fact that the assembly process may adapt to the failure of any application component, but will fail utterly should the assembly process itself encounter a problem. The solution to these problems is to do away with the central node and to use a fully distributed technique.

The distributed approach described hereafter assumes that the system comprises a set of peer nodes (physical hosts connected in a network), each of which knows only the components that it hosts locally. In particular, the dependency graph cannot be computed without examining every node. Each component knows some (possibly empty) subset of the structural constraints, knows all the functional requirements, and knows the NF annotations of the local components and the NF preferences. The objective is to have the peers derive and agree on a (global) configuration given only their restricted (local) knowledge. To avoid the performance limitations encountered in the approach of Georgiadis *et al.* [GMK02], we would like an approach which tolerates temporary inconsistency between the views each node has, provided that eventual agreement is guaranteed.

5.1 Gossip

A *gossip* protocol provides an ideal means to achieve global agreement using decentralised information. Gossip protocols are used to ensure that a network of peers all receive some piece of information within a certain time, without resorting to reliable broadcast, which may involve a huge number of messages (restricting scalability). In our case the item of information is the (global) component configuration. There are two kinds of gossip protocol: the basic ones which involve a single update (which we shall call *simple*), and *aggregate* protocols, which, through the propagation of several updates, compute some aggregate function of information stored at each node. Our protocol is of the latter kind.

5.1.1 Simple Gossip

Gossip was originally inspired by the way in which disease epidemics spread across a population [DGH+87]. In its simplest form, gossip is a protocol for propagating a single piece of information (a database update for example [DGH+87]) across a network. Initially one node has the information (which we refer to as the *state*), and chooses another node at random to transmit the information to. The protocol proceeds in synchronous rounds so that in the next round, both the informed nodes independently choose another node to update. In the best case, four nodes now have the information (Figure 5.1(a)), but since the choices are uniform and independent, the same nodes may have been chosen again, leaving the same two informed nodes (Figure 5.1(b)). If all nodes continue transmitting indefinitely, the protocol is referred to as *anti-entropy*, and the information is guaranteed to reach all nodes in the network [DGH+87]. If nodes decide to stop transmitting after some number of rounds, the protocol is referred to as *rumour mongering*.

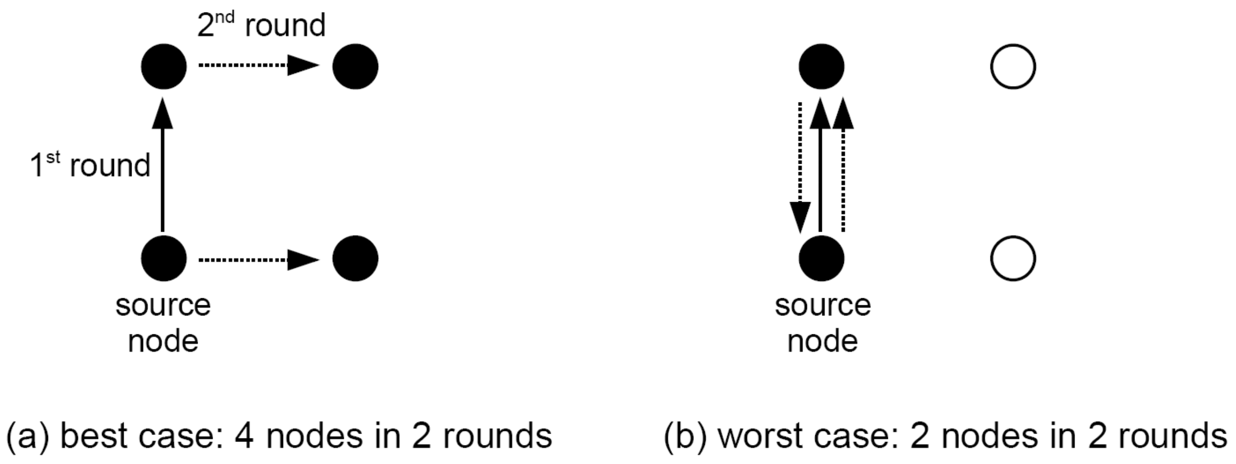

2nd round
1st round
source node
source node
(a) best case: 4 nodes in 2 rounds
(b) worst case: 2 nodes in 2 rounds

Figure 5.1: Best-case and worst-case instances of simple gossip

Variations of the anti-entropy protocol can be created by adjusting the strategy for selecting nodes (round robin is one alternative) or by performing extra steps when nodes communicate. With a *push* policy, the update is transmitted from the choosing node, while with a *push-pull* policy, updates pass in both directions. In the uniform push (anti-entropy) algorithm, a single state update will propagate across the network in $\log_2 n + \ln n + O(1)$ rounds, where n is the size of the network [DGH+87, Pit87]. Intuitively, this result stems from the fact that the fastest propagation would require every node to push the state to a "new" node in every round. This would mean that the number of informed nodes doubles in every round, and thus the number of steps required to reach n is $\log_2 n$. However, the random selection of nodes means that propagation is slower than optimal. Nevertheless, the speed of propagation ensures that gossip scales well: for 1000 nodes, dissemination of a single update may take as few as 17 rounds.

The random pattern of message exchange ensures that the loss of a single message only delays agreement since the node that would have received the update will eventually receive another message. Likewise if a node fails and restarts, it only has to wait for a single message to arrive to restore its state.

5.1.2 Aggregate Gossip

In an *aggregate* protocol [KDG03], each node is permitted to compute some function of the received state and use the result of this function for propagating to other nodes. This function may incorporate some information known only to that node (which is never transmitted directly). This behaviour allows the protocol to compute some aggregate function of the information held by each node. For example, if every node holds an integer, the nodes can agree on the maximum value of these integers using an aggregate protocol [JMB05]. This is achieved by treating the information propagated as an estimate of the maximum, and by having every node keep a local estimate (set to the value of the node's integer in the first step). Each node then compares the received estimate with its own estimate and chooses the largest to produce a new estimate of the maximum. As estimates propagate across the network, the estimate of each node will converge on the true value of the maximum, and every node will be aware of it.

Computing the maximum sometimes requires a node to reveal its own integer (when the new estimate is that value), but other aggregate functions can completely hide the information held by each node.

An *aggregate function* (one which converges under these circumstances) must (i) be divisible into pairwise operations (so a new estimate can be computed from an old estimate and local information) and (ii) proceed monotonically towards the true value (according to some distance function) [vR].

In the case of computing a maximum, this is easily divided into pairwise operations. It is also trivially monotonic since (i) if the received estimate is greater than the node's estimate, the new estimate is the same and (ii) if the node's estimate is greater, then the new estimate is greater than the old estimate, and so must be closer to the true maximum. If the aggregate function is non-monotonic (such as if the node alternated between computing a minimum and a maximum) then the estimate will never converge.

5.2 Gossip-Based Assembly

Our distributed assembly process uses an aggregate gossip protocol to enable the set of nodes to derive and agree upon a global component configuration. Each node may host some subset of the components in the configuration, and only a subset of the nodes in the network may be involved in hosting components. Once a configuration is agreed upon, it is instantiated by determining the wiring automatically as described in Section 4.2.3. If a component failure occurs, the gossip protocol resumes to find a new solution for the functional requirements.

The gossip protocol for assembly uses a uniform push policy. Each node's state is an "estimate" of the (global) component configuration, and can use the local component repository to propose a new state. Figure 5.2 shows an overview of the process in which two nodes exchange a single gossip message, resulting in a change of state. In the following the state representation (and thus the format of gossip messages) is given before describing the rules used to generate a new state.

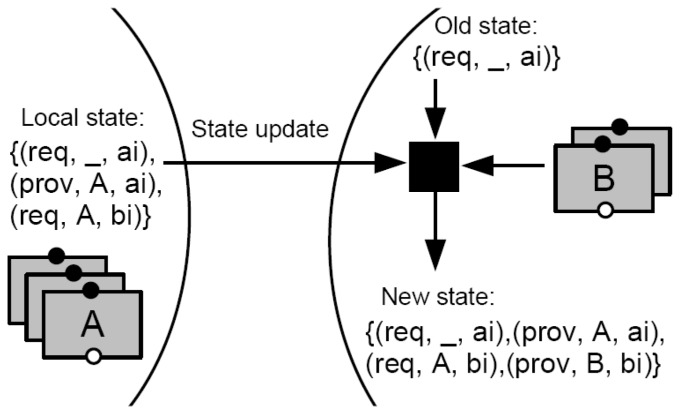


Figure 5.2: Overview of gossip-based assembly

5.2.1 State Representation

The state of a node is an estimate of the global component configuration. The state is a set of “active” dependencies which represent a decision on which component provisions will satisfy which requirements. Gossip messages contain a single state.

$$\begin{aligned} \mathit{State} &\subseteq 2^{\mathit{Dependency}} \\ \mathit{Dependency} &\subseteq \{\mathbf{prov}, \mathbf{req}\} \times \mathit{ComponentType} \times \mathit{InterfaceType} \end{aligned}$$

A *Dependency* is a tuple of a *ComponentType*, an *InterfaceType* and **prov** or **req** indicating whether the dependency represents a provision or a requirement. In the case of a provision $(\mathbf{prov}, c, i)$, a component of type c is present in the configuration and is responsible for providing interface i. A $\mathit{ComponentType} \subseteq \mathit{ImplementationType} \times \mathit{NodeIdentifier}$ is in fact a pair comprising the actual implementation type and the node on which the component is hosted. This way, multiple implementations of a type are supported, provided they reside on different nodes.

Given a *State*, the set of component instances active in the configuration can be found by collecting the *ComponentType*s mentioned in each *Dependency*. For example, the state

$$\{(\textbf{req}, (c, n_0), i), (\textbf{prov}, (d, n_0), i), (\textbf{req}, (d, n_0), j), (\textbf{prov}, (e, n_1), j)\}$$

describes a configuration of $\{c, d\}$ hosted on node n_0 and e hosted on n_1. Component d satisfies requirement i of c and requires j, satisfied by e. The wiring of the configuration is determined automatically as described in Section 4.2.3.

5.2.2 State Transformation Rules

Upon receipt of a new state s, the node applies the following rules:

1. If the state does not contain one of the functional requirements i, then the node adds $(\textbf{req}, \_, i)$.
2. If there is a requirement $(\textbf{req}, (c, n_0), i)$ with no corresponding provision $(\textbf{prov}, (d, n_1), i)$ and the node n knows a component e which provides interface i, then the node can add $(\textbf{prov}, (e, n), i)$ to the state along with the other provisions and requirements of e. This rule leads the configuration towards being complete[1]. If there are multiple local providers of i, non-functional preferences are used to choose between them.
3. If the above rules do not apply, the state is complete and meets the functional requirements, and the node may evaluate it against the structural constraints using a local constraint checker. If the check fails, the node can
   (a) adopt a randomly-selected previous state (for this purpose, a list of all incomplete states is maintained), which simulates the backtracking of the centralised algorithm, or

[1]A symmetric rule which removes provisions that are not required is also possible, and ensures minimality. It is however unnecessary as the configurations generated are minimal in any case.

(b) add a randomly-selected component to the state, which simulates the exhaustive search of the centralised algorithm.

The new state s' is only accepted if the following conditions hold: (i) s' is not a subset of the node's previous state s_n, (ii) s' must be complete and valid if s_n is complete and valid, and (iii) $s' > s_n$ according to the partial order $>$ (we use a lexical ordering over component names). Conditions (i) and (ii) ensure that the algorithm is monotonic: a node cannot accept an incomplete solution when it already knows a more complete solution. Condition (iii) ensures that when there are two possible complete solutions, every node chooses the same one.

In the next section we illustrate the essential features of the protocol before addressing some of the finer points necessary to complete the description.

5.2.3 Example

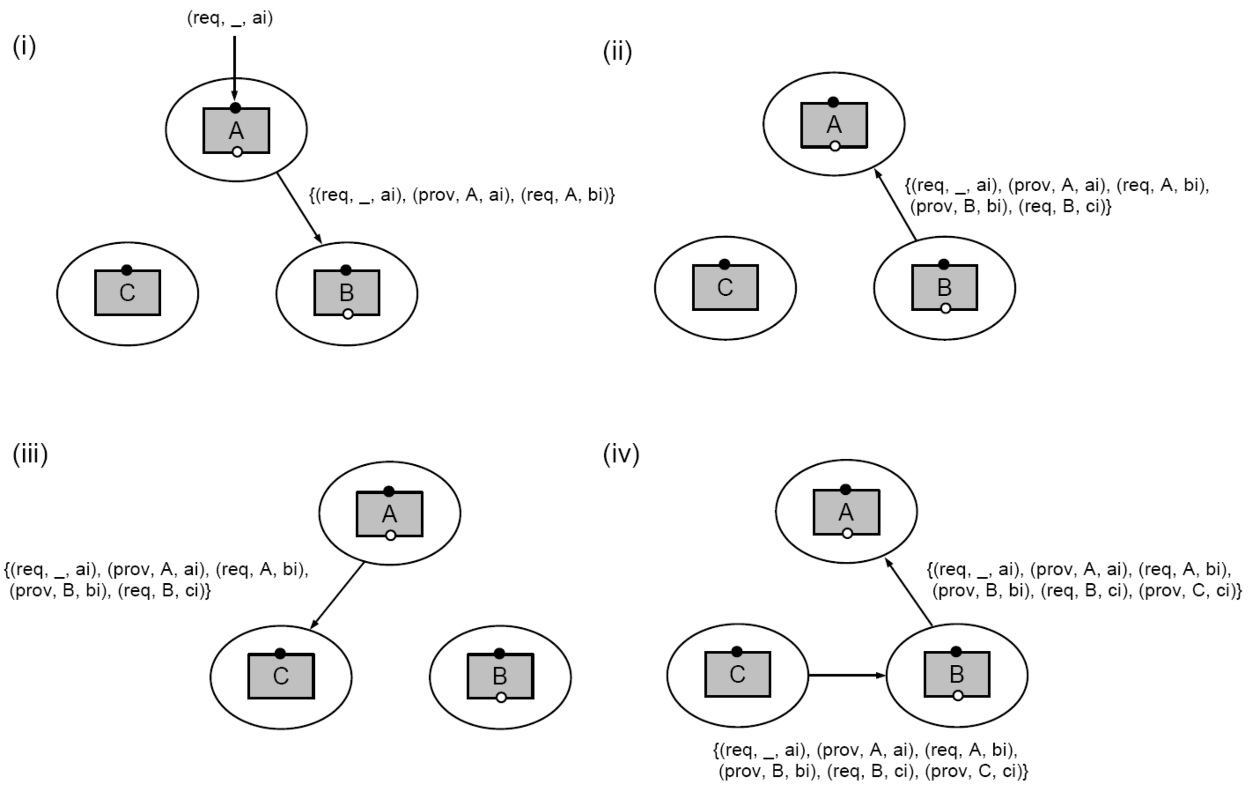

(i)
(req, _, ai)
A
{(req, _, ai), (prov, A, ai), (req, A, bi)}
C
B
(ii)
A
{(req, _, ai), (prov, A, ai), (req, A, bi), (prov, B, bi), (req, B, ci)}
C
B
(iii)
A
{(req, _, ai), (prov, A, ai), (req, A, bi), (prov, B, bi), (req, B, ci)}
C
B
(iv)
A
{(req, _, ai), (prov, A, ai), (req, A, bi), (prov, B, bi), (req, B, ci), (prov, C, ci)}
C
B
{(req, _, ai), (prov, A, ai), (req, A, bi), (prov, B, bi), (req, B, ci), (prov, C, ci)}

Figure 5.3: Example of distributed assembly using gossip

Figure 5.3 shows a system composed of three nodes each of which hosts one component, A, B or C (we shall use these names to refer to both the node and the component, using italic type to distinguish the nodes). The interface ai is identified as a functional requirement and is provided by A. A also requires bi. Thus in step (i) node A identifies a requirement it can satisfy and adds the provision $(\mathbf{prov}, \mathrm{A}, ai)$ and the requirement $(\mathbf{req}, \mathrm{A}, bi)$ to the state. A transmits this new state to a randomly-selected node, which in this case is B. Likewise, in step (ii), B sees that $(\mathbf{req}, \mathrm{A}, bi)$ is a requirement which it can satisfy. Component B has the further requirement of ci, so $(\mathbf{req}, \mathrm{B}, ci)$ and $(\mathbf{prov}, \mathrm{B}, bi)$ are added to the state. This is transmitted back to A. In step (iii), A makes no changes to the state (since ci cannot be satisfied locally) and at some point forwards it to C. In step (iv), C satisfies the requirement for ci by adding $(\mathbf{prov}, \mathrm{C}, ci)$ to the state, and forwarding it arbitrarily to B. B has no changes to make and eventually forwards it to A. After step (iv) all nodes agree on the state $\{(\mathbf{req}, \_, ai), (\mathbf{prov}, \mathrm{A}, ai), (\mathbf{req}, \mathrm{A}, bi), (\mathbf{prov}, \mathrm{B}, bi), (\mathbf{req}, \mathrm{B}, ci), (\mathbf{prov}, \mathrm{C}, ci)\}$ which gives the configuration {A, B, C}.

Since the configuration after step (iv) is complete, it can be evaluated against the user's structural constraints. Any node can perform this evaluation, and in the case of failure revert to some previous state. Unfortunately in this case there are no alternative configurations so the search would fail.

5.2.4 Enforcing Structural Constraints

Unfortunately, deriving a complete configuration using rules 1 and 2 as in the example does not guarantee that the solution will meet the structural constraints, which can only be checked when the solution is complete. Therefore it is necessary to extend the approach beyond an ordinary aggregate protocol to incorporate backtracking when a complete solution fails the constraint check. This is the purpose of rule 3, which gives the node a choice between (a) backtracking to a previous state and (b) adding an arbitrary component.

In case (a), the node chooses at random an entry from the list of all states it has seen previously, and propagates it. Since this state is likely to be an “ancestor” which led to the derivation of the invalid configuration, it is necessary to prevent the derivation of the invalid configuration by informing the other nodes that it fails the constraint check (in case they have not realised themselves). For this purpose we have adapted the “death certificates” of [DGH+87] into a secondary gossip protocol. When a state fails the constraint check, a death certificate is produced and propagated across the network using a uniform push policy. Upon receiving a death certificate, the state mentioned is added to a local list and if the offending state ever recurs, its propagation is suppressed.

The propagation of death certificates forces the algorithm to choose alternative component implementations (by, for example, overriding the $>$ ordering on solutions). Since backtracking is random, eventually either a valid solution will be found, or every node will contain a list of death certificates for every possible configuration.

Notice that death certificates are not strictly necessary if every node has the full set of structural constraints, since every node would build up the list of death certificates locally without any propagation. However, in that case death certificates would work as an optimisation since they can be issued before the network has agreed upon a solution, allowing nodes to dismiss unseen solutions. In other words, using death certificates takes advantage of parallelism in that different solutions can be checked simultaneously by different nodes, as opposed to requiring every node to check every solution.

Case (b) simulates the exhaustive search of the space of components which is performed in the centralised algorithm when no solution in the transitive closure of the dependency relation satisfies the constraints. Again, random choice ensures that eventually a valid solution is found or the full combinatorial space is covered.

5.2.5 Convergence & Non-Convergence

There are two properties of interest when considering convergence (agreement on a single solution). The first (and most important) is whether the algorithm is guaranteed to converge in all cases. The second is how quickly one can expect convergence to occur, with respect to the size of the problem and the number of nodes involved.

In order to show that convergence is guaranteed, we need to show

(i) that the aggregate protocol proceeds monotonically to a solution and

(ii) that all nodes come to agreement on that solution.

Firstly consider the protocol without backtracking. This restricts our view to rules 1, 2 and 3(b) of Section 5.2.2. Notice that each of these rules can only add $Dependency$ entries to the state, and so the state can only increase in size. Additionally, if a node receives two updates "out of order" whereby the second update is an ancestor (a subset) of the first, only the first is retained. This means the state cannot decrease in size (unless it is a different solution entirely, in which case we rely on gossip to achieve agreement). Hence, the protocol is monotonic, giving (i).

For (ii), we rely on standard properties of gossip that allow us to assume every node will eventually receive every update (that is not rejected by another node). Then we must ensure that every node chooses the same solution from this "acceptable" set. It is for this purpose that we order solutions using $>$, and have nodes choose the greatest. This leads to agreement, giving (ii).

The effect of (i) and (ii) is a protocol in which the various solutions grow monotonically until completion, at which point each node makes a choice using the arbitrary (but fixed) ordering. However, so far we have omitted backtracking. Clearly backtracking breaks monotonicity, so it is necessary to show that backtracking does not lead to infinite loops. Every time backtracking occurs, a death certificate is produced for the invalid configuration. This is propagated across the network using uniform push (and so all nodes eventually have the death certificate). Since backtracking is performed by making a random choice between the previous states (and all solutions are reachable from the "root" state) we can say that if there is a valid solution, it will eventually be derived (since the probability of backtracking to the necessary state is non-zero). If there is no valid solution, then every node will receive a death certificate for every possible solution, and backtracking will continue indefinitely. A time-out must be used in this case since although a node may have a death certificate for all solutions, there is no way for the node to detect this situation, since the node does not know the whole dependency graph. The unfortunate side effect of a time-out is that the solution may be found just after the time limit, and so the algorithm is no longer complete (in the sense of being able to derive solutions when they exist). Depending on the application domain, it may be a better policy to fall back to the centralised algorithm after a time-out to ensure completeness.

In summary, without any structural constraints the algorithm is guaranteed to converge on a single solution, even if that solution is incomplete because of a requirement that cannot be satisfied. When structural constraints cause backtracking, the algorithm will find a solution if one exists and if it is found before the time-out.

5.2.6 Time To Convergence

In a uniform push protocol (and in several aggregate protocols [KDG03]), convergence after an update occurs in $O(\log n)$ rounds [DGH+87]. Assembly, if backtracking is disregarded for the moment, can be seen as several such updates performed (in the worst case) sequentially. Then for a solution size (number of components[2]) s, convergence can be expected within $O(s \log n)$ rounds. Ordinarily, of course, updates happen in parallel, providing even better performance.

5.2.7 Detecting Convergence

Although the protocol theoretically converges within a certain number of rounds, a mechanism is required for individual nodes to detect that this is truly the case. The algorithm has converged when

- **C1.** every node has the same state, and
- **C2.** no node can suggest a new state according to rules 1 and 2.

This situation can be detected by each node independently when

- **L1.** observes a lack of state changes for $O(\log n)$ rounds. If no node can suggest any changes, then the time remaining before convergence is the time it takes for a single update to propagate in a non-aggregate push algorithm, which is $O(\log n)$, and
- **L2.** cannot apply rules 1 and 2 to the current state.

It is trivial to see that if condition L2 holds on all nodes, then C2 holds. To see how condition L1 ensures C1, suppose that C1 does not hold (but C2 does). Then the nodes disagree on the state but (since C2 holds) they cannot suggest any changes. In this case, the nodes will come to agreement within $O(\log n)$ rounds (under the normal expectations of gossip). Thus if C1 does not hold, a node can expect a state change to occur within $O(\log n)$ rounds, falsifying L1.

When both L1 and L2 hold, a consensus protocol is used to ensure that C1 and (particularly) C2 hold. For this purpose, we use *decentralised two-phase*

[2]Strictly the solution size is the number of *Dependency* entries but this is usually proportional to the number of components.

commit [Lyn96], whereby every node transmits its vote (stating whether L1 and L2 hold) to every other node (requiring full connectivity). When each node has received a “converged” vote from every node, the node can be sure that C1 and C2 hold and thus the algorithm has converged. This relies on the *weak AC* (atomic commitment) property of two-phase commit, namely that, in the absence of node (crash) failures and message loss, all nodes eventually decide either that they have converged (when all the votes agree) or that they have not converged (when some vote disagrees).

Decentralised two-phase commit is a convenient choice because of its simplicity and its negligible effect on performance, adding only two rounds to the time if it succeeds on the first attempt. However, it requires a large number of messages to be sent ($O(n^2)$), and could be replaced by another scheme such as centralised two-phase commit (which requires fewer messages) or three-phase commit. Another alternative would be to use the variant of gossip known as rumour mongering (Section 5.1.1) whereby, with a certain probability, nodes stop propagating an update after a certain number of rounds. Eventually, all nodes will stop without having engaged in any extra commit protocol. However, this benefit must be balanced against the small chance that updates will not reach every node, causing nodes to have inconsistent solutions [DGH+87].

5.3 Simulations

In order to test our hypotheses, particularly with respect to the performance of the distributed assembly algorithm, we have performed several simulations using, in the first instance, predictable graphs of components (with and without backtracking) and, in the second instance, random graphs of components.

5.3.1 Heterogeneous Rings

In the first simulation of regular structures, the sets of components were structured into rings of increasing sizes. Each component was of a different type, providing and requiring a single interface (in other words, the ring structure was built into the dependencies rather than enforced as a constraint). There were no further dependencies, nor structural constraints, such that there was only a single trivial solution in each run (the number of components c equals s, the solution size). The assignment of components to nodes was also made predictable. Figure 5.4 shows a ring of size 4.

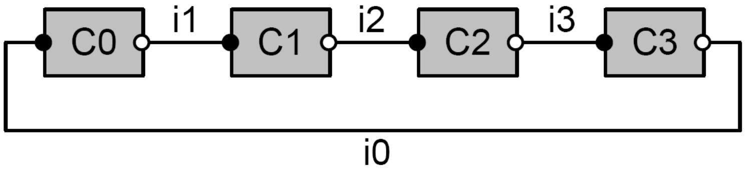


Figure 5.4: A heterogeneous ring

Figure 5.5 shows the behaviour of the algorithm given a ring comprising 20 components, and increasing numbers of nodes. This fits a logarithmic curve, confirming our expectation of convergence within $O(s \log n)$ rounds. Notice that the best fit curve has a coefficient of approximately 20, matching our expectation that a solution of size 20 will require at most 20 sequential updates. The final term in the formula can be regarded as the start-up time, and if divided by 20, this is less than 1 round.

Figure 5.6 shows the behaviour with a fixed number of nodes, and increasing sizes of ring. The relationship is linear as expected ($O(s \log 5)$). The number of rounds is approximately double the size of the ring.

Figure 5.7 shows the number of rounds when the size of the ring is fixed to the number of nodes, and increased. The expected result is that the number of rounds taken to converge is $O(n \log n)$. The results in this case should grow slightly faster than linear, but this is difficult to confirm using the graph.

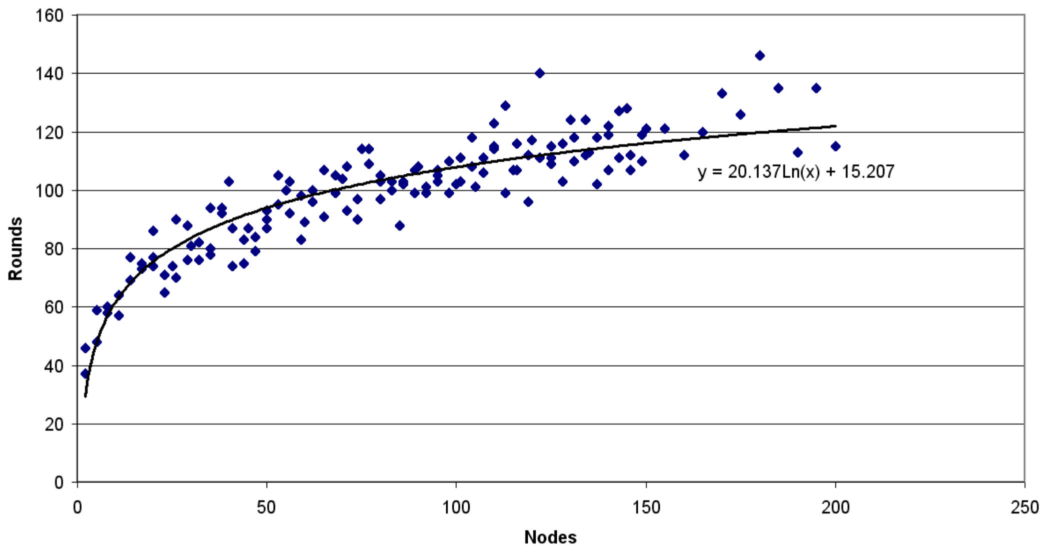


Figure 5.5: Simulation for a ring of size 20, with increasing nodes

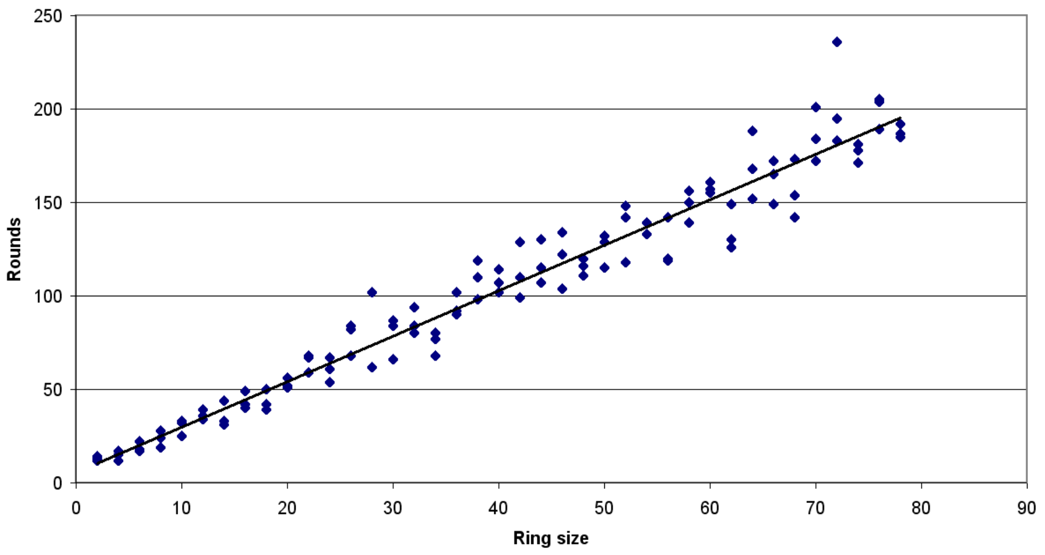


Figure 5.6: Simulation with 5 nodes, and increasing sizes of ring

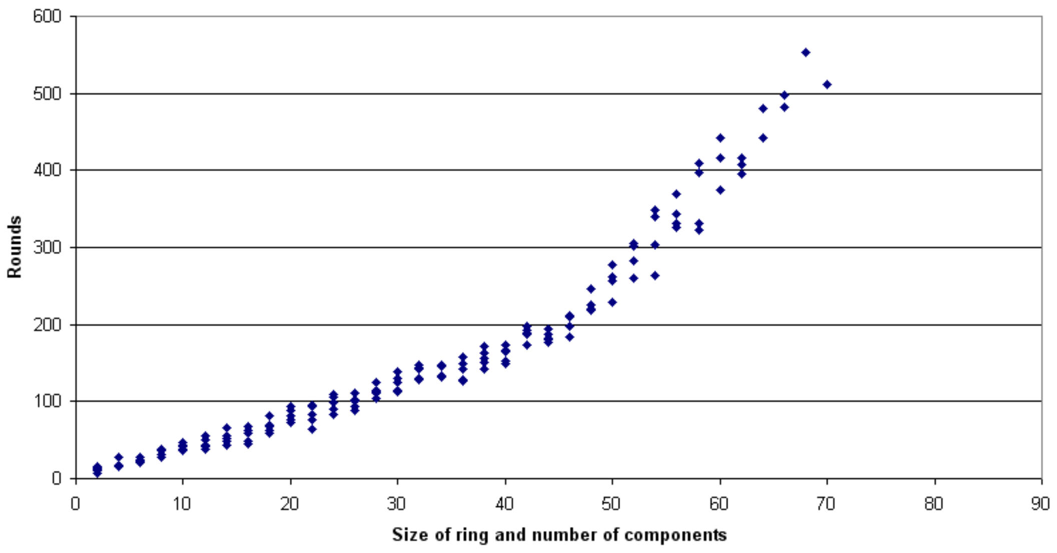


Figure 5.7: Simulation with increasing sizes of ring, matching the number of nodes ($n = s$)

5.3.2 Homogeneous Rings

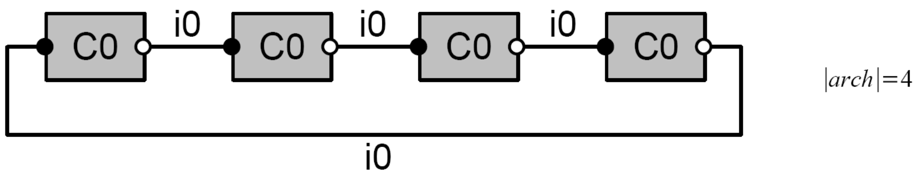


Figure 5.8: A homogeneous ring

A more common and expressive way to enforce a regular structure (such as an architectural style) is through the use of structural constraints. In the next set of simulations, a single component type is used, and the ring structure is enforced through the constraints. In order to construct a ring of a certain size, composed of multiple instances of the same component type, it is necessary that several different nodes host the component (recall that each instance must be on a different host). Also, since the size of the ring bears no relation to the functional requirements (which are satisfied by a single component), exhaustive search (that is, rule 3(b) of Section 5.2.2) must be used to find the combination of components which satisfies the constraints. Figure 5.8 shows a ring which is constrained to have 4 elements.

Figure 5.9 shows the number of rounds to generate homogeneous rings of 3

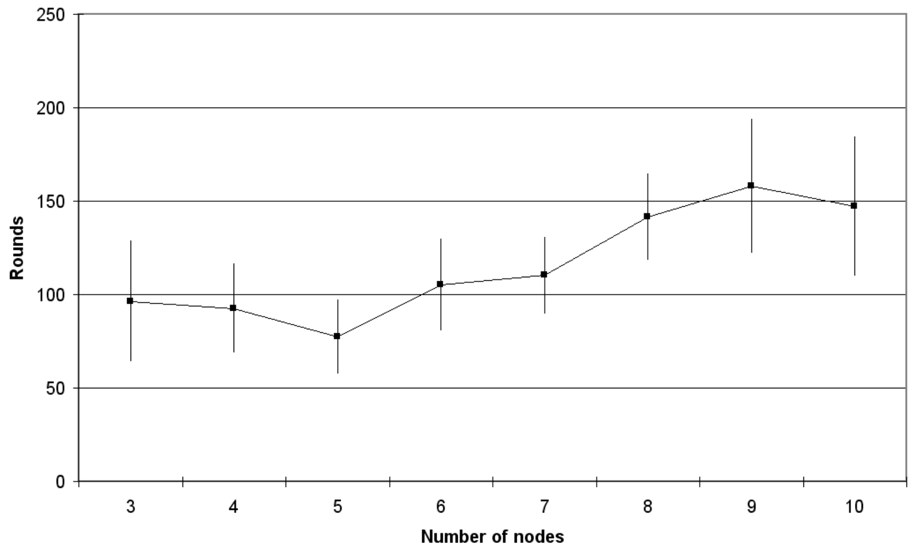


Figure 5.9: Simulation with increasing numbers of nodes for rings of size 3

components, for increasing numbers of nodes. In this case the graph shows the mean average rather than the results of individual runs. This is due to the much wider spread of points resulting from the random behaviour of backtracking. The vertical bars on each point show the 95% confidence interval [Dev95] which gives an indication of how reliable each mean is. The graphs shows that the average number of rounds increases slowly with the number of nodes. However, for numbers of nodes greater than 10, simulation becomes infeasible as the execution time for a single run exceeds our time limit of 120 seconds. This occurs because the execution time grows faster than the number of rounds, due to increased computation in each round (arising in part from the growing number of death certificates). Since we are simulating many nodes on a single computer, these costs are sequential rather than parallel.

Figure 5.10 shows the increasing number of rounds taken to generate homogeneous rings of various sizes in a network of 5 nodes (limiting the maximum ring size to 5). Again, the graph shows an average with a 95% confidence interval for approximately 200 runs (in total). The increase in the number of rounds can be understood by considering the likelihood of choosing a valid solution at random. Since there are 5 nodes which can each host a component instance, there are

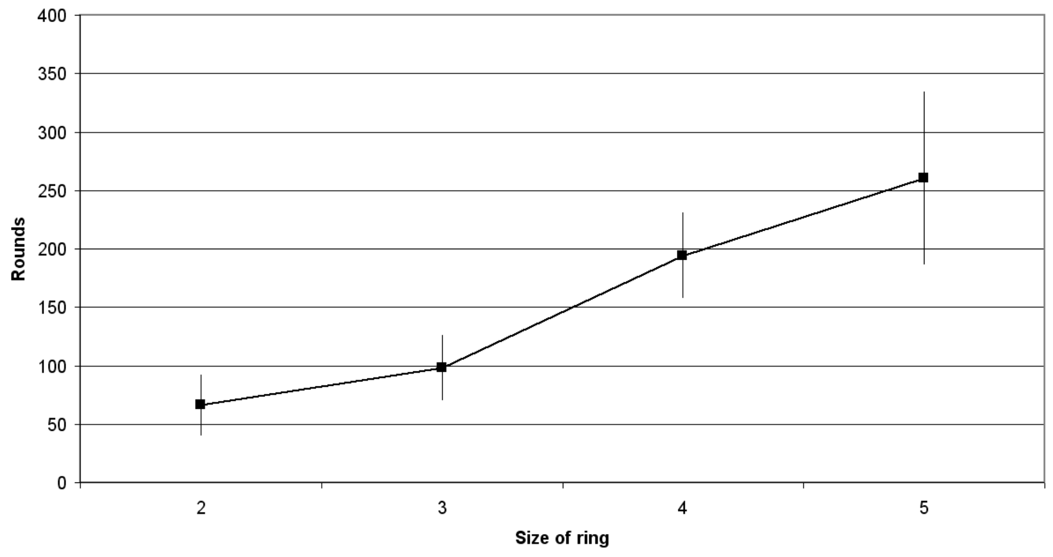


Figure 5.10: Simulation with increasing sizes of ring, with 5 nodes

$2^5 = 32$ solutions, some valid and some invalid. When the ring is constrained to be of size 2 or 3, there are $\binom{5}{2} = 10$ valid solutions. The chance of picking a valid solution is then $\frac{10}{32} = 0.31$. For rings of size 4 and 5, the number of valid solutions falls to 5 and 1 respectively, giving probabilities of 0.16 and 0.03. This means that the number of rounds taken to reach a valid solution are likely to be greater for these larger sizes of ring, although in practice death certificates ensure that solutions are not attempted twice, effectively increasing the probability of choosing a valid solution as the algorithm progresses.

5.3.3 Random Components

The next set of simulations use sets of randomly-generated components with no structural constraints. The success of the algorithm on random components shows that it is not tuned to specific styles such as rings, to configurations of a certain size, nor does it rely on the intuition of the designer, in the way that repair strategies are written to handle a very restricted domain of components.

Given a number c of components, the number i of interfaces was set (randomly) between 25% and 50% of c. Having a very large i reduces the chance that the algorithm can find a solution, since it is less likely that a component provides a given interface. Then, for each interface, a port providing or requiring that interface was added to each component with probability $\frac{k}{c}$ with k set to 4. Assigning an interface to a single component at random would mean that assignment to a particular component has a probability of $\frac{1}{c}$. Multiplying this probability by 4 causes each interface to be assigned to 4 components, on average. For the same reason, each component has an average of 4 ports. A small k, particularly $k < 1$, reduces the chance that a solution can be found. The choice between provisions and requirements was made equally likely. An extra parameter d controls how many duplicates of a component exist. When this is zero, each component is hosted by a single node. When this is non-zero, d duplicates are assigned to nodes at random.

The first simulation in Figure 5.11 shows the number of rounds before convergence was detected for increasing numbers of nodes over several runs. 40 randomly-generated components were used as input, with $k = 4$ and $d = 0$.

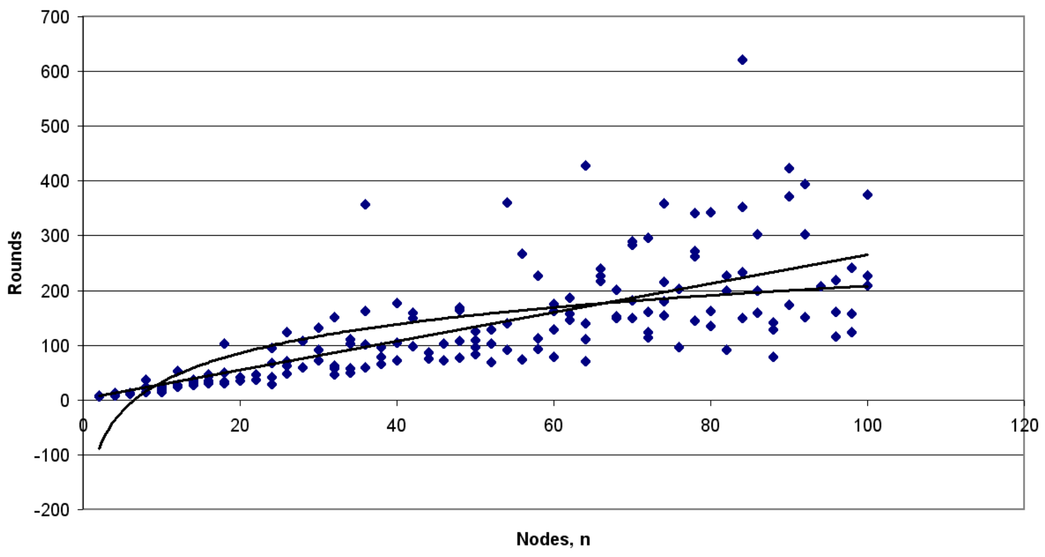


Figure 5.11: Simulation for 40 random components

In this case it is hard to say whether a linear or logarithmic curve fits the results best. Convergence is expected in $O(s \log n)$ rounds, but each run (each point in the graph) has a different s ($1 < s \leq 40$) due to the fact of random generation. This would explain a "noisier" curve than with regular structures. In any case the graph shows that the number of rounds grows slowly with the number of nodes.

In the second simulation, we increased the number of components while keeping the number of nodes n fixed at 5. The other parameters were as before. We expected that the number of rounds would, in the worst case, grow linearly with the solution size s. Figure 5.12 shows the convergence time against the total number of components. Here, larger sets of components show a slightly larger number of rounds to converge. This was expected since the solution size is likely to increase given a larger set of components, so the convergence time should increase also.

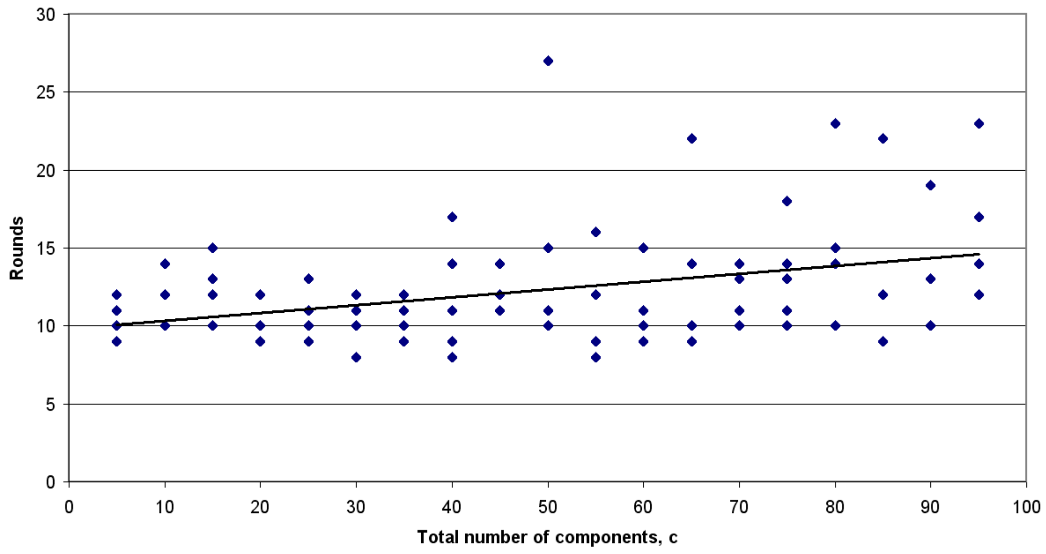


Figure 5.12: Simulation for increasing numbers of components, with 5 nodes

Figure 5.13 shows that, indeed, the number of rounds does increase slowly with the solution size.

The final set of random simulations shows, for the sake of completeness, the effect of varying the parameters which are used to produce the random components.

Figure 5.14 shows the effect of increasing the parameter k. The other parameters were fixed as above. A greater k leads to a greater average number of ports per component, and a greater average number of implementers of each interface. This consequently leads to a greater solution size s, as the dashed best fit line shows

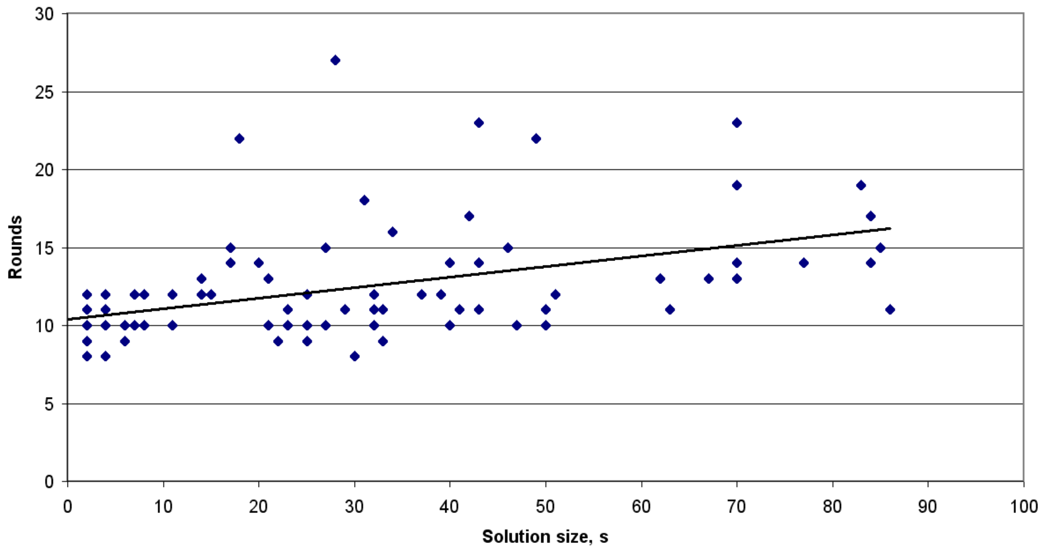


Figure 5.13: Simulation with 5 nodes, and increasing solution sizes

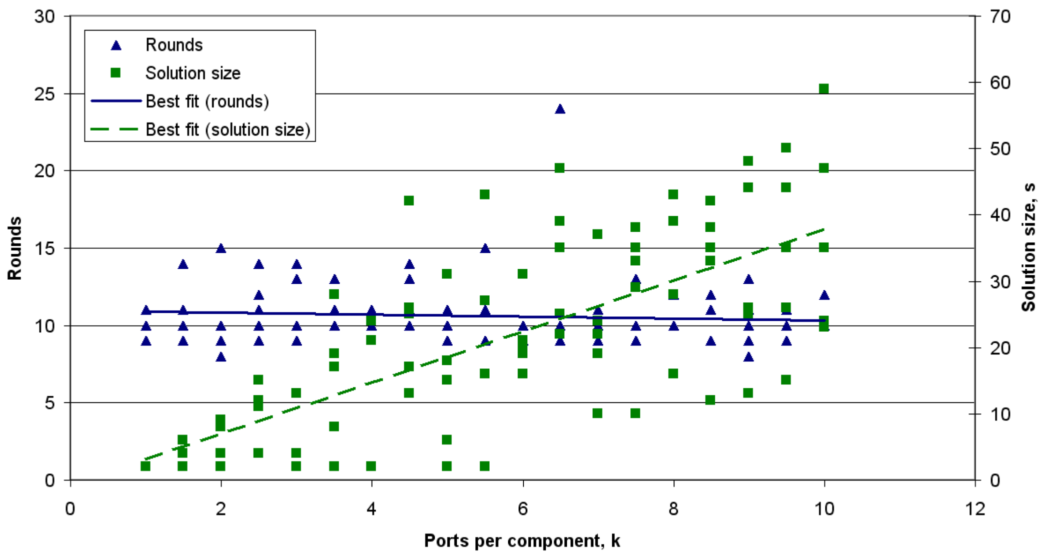


Figure 5.14: Simulation for 20 components over 5 nodes, with variable k

(this line relates to the second vertical axis). The number of rounds to convergence is shown to be almost constant. This was expected to increase slightly with the solution size, which further data may show.

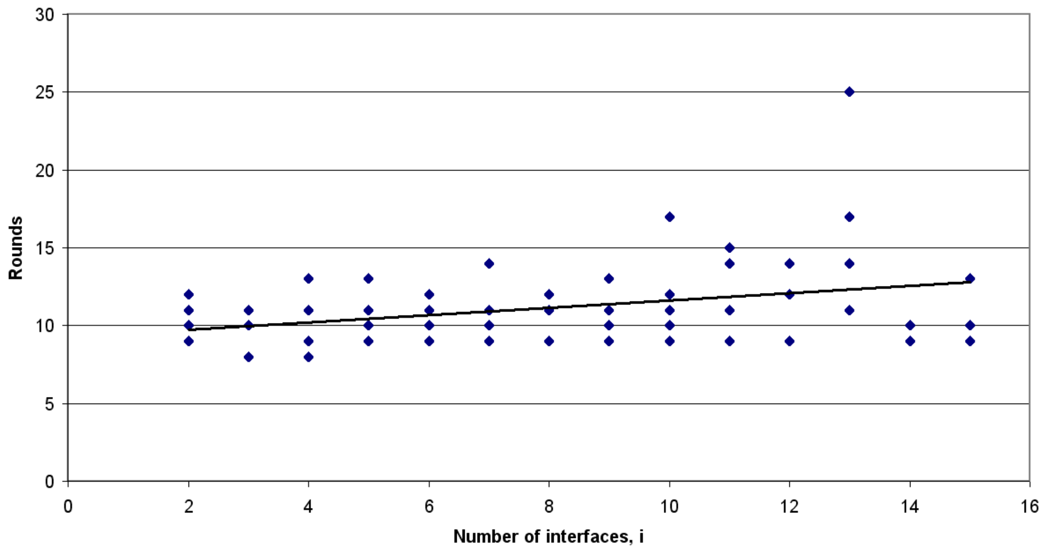


Figure 5.15: Simulation for $c = 20$, $n = 5$, $k = 4$, with variable i

Figure 5.15 shows the effect of increasing the number of interfaces i between 2% and 75% of c ($c = 20$). k was held at 4. Again, an increased i leads to a larger solution size, which leads to a slight increase in the number of rounds.

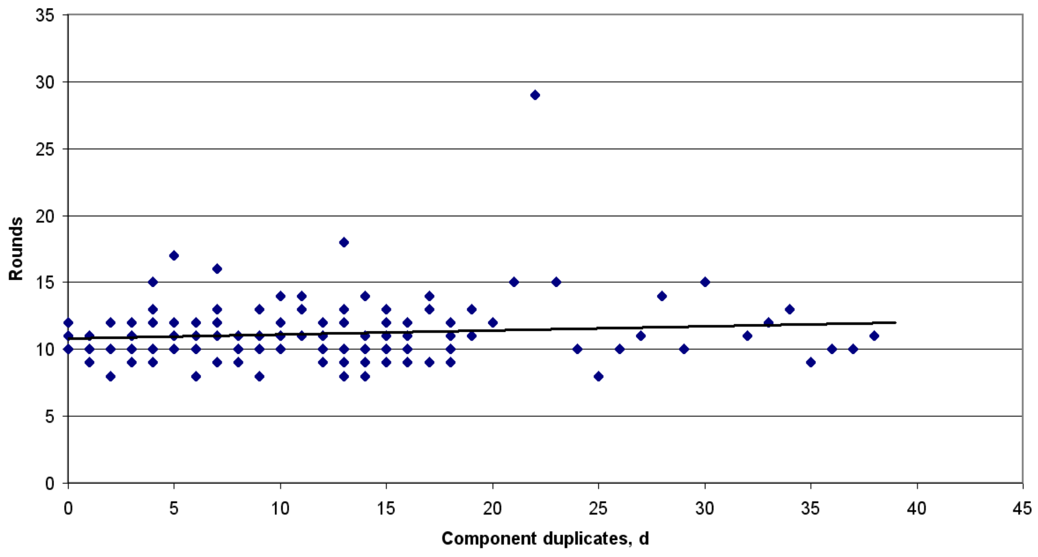


Figure 5.16: Simulation for $c = 20$, $n = 5$, $k = 4$, i proportional to c, with variable d

Finally, Figure 5.16 shows that component duplication has little effect on the number of rounds to converge.

5.3.4 Message Loss

The final simulation in Figure 5.17 shows how the protocol copes with message loss. Here, the simulation discards varying proportions of gossip messages from 10% to 85% and the time to convergence is measured. No convergence messages were discarded because this would prevent convergence under two-phase commit. The protocol tolerates up to 50% message loss with a small increase in the number of rounds, but this increases sharply such that 85% message loss causes a four- to five-fold increase in the time to convergence.

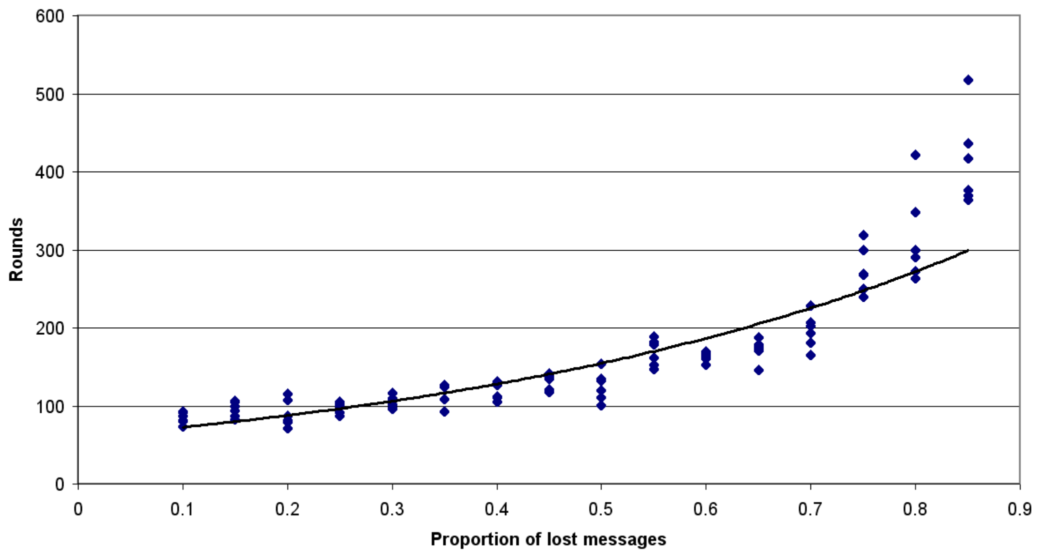


Figure 5.17: Simulation of the effect of increasing message loss, on a ring of size 20 with $n = 5$

5.4 Optimisations

There are a number of optimisations one could consider applying without changing the fundamentals of distributed assembly using gossip. Here we discuss the feasibility and benefits of three such optimisations.

5.4.1 States As Deltas

The described approach contains a lot of redundancy in the gossip messages, wherein the entire state is transmitted, even when most of the state has been agreed upon. The size of each message is (approximately) proportional to the size of the solution s, and over $O(s \log n)$ rounds, the total "message volume" (per node) can be expected to be $O(s^2 \log n)$.

An optimisation, then, is to restrict the gossip messages to transmitting the change from the previous state (the *delta*). For example if a node satisfies a requirement r by adding component c, it will transmit (and other nodes will propagate) the delta $\{(+, \textbf{prov}, c, r)\}$. A delta may contain an addition denoted by $+$ or a removal denoted by $-$.

Upon receiving a delta, a node applies the delta to its current state (by adding or removing the indicated dependencies), and propagates the delta. If the node can apply rules 1-3, it transmits a further delta describing its changes. The effect of the optimisation should be to reduce the message volume by a factor of s, giving $O(s \log n)$.

However, there is a problem with this apparently fruitful adjustment since a delta can be applied when the states of the sender and receiver disagree. In such circumstances the delta may not even be relevant, satisfying a requirement which does not yet exist in the recipient state. This can be the result of a provision delta arriving before a requirement delta, in other words, when deltas arrive "out of order". This was previously avoided by virtue of the fact that the requirement would be included in the full state transmitted when the provision was added (and the subsequent arrival of a message with the requirement but not the provision was ignored).

The effect of messages arriving out of order is that the states of the sender and receiver can diverge (through application of rules 1 and 2 to already differing states) rather than converge as the original approach would have ensured.

Moreover, when alternatives exist for a requirement, all alternatives are received and applied, creating a much larger solution than is necessary. This is because they can no longer be distinguished as alternatives where previously nodes had to choose one of the alternative states. If we attempt to distinguish between them on the basis of which requirements are being satisfied, we lose the ability to have multiple implementations of an interface and multiple instances of a component type within a configuration.

Properly addressing these issues would require some form of global order on deltas and a relation between deltas and the states to which they can be applied. This difficulty, and the modest saving in message volume suggest that the optimisation is not justified. However, in order to demonstrate the benefit the optimisation might give if adopted, we tested the algorithm with a rudimentary measure that requires a node to fall back to transmitting the full state (to all nodes) when the node has found a complete and valid solution. This full state is adopted by the recipients, overriding any other solutions. Clearly this increases the message volume somewhat (though it does not seem to negate the benefit), and more importantly, runs counter to the aims of gossip.

Figure 5.18 shows the benefit of the optimisation, which is a reduction in the total message volume. This simulation was performed on a heterogeneous ring of size 20. The delta optimisation fits a slowly increasing logarithmic curve, while the original algorithm grows rather more quickly.

Figure 5.19 shows that the delta optimisation has no discernible effect on the number of rounds to generate rings of size 20, since this graph is almost identical to Figure 5.5 (without the optimisation). Likewise, if the number of nodes is fixed to 5 and the size of the ring is varied, a linear graph similar to Figure 5.6 emerges.

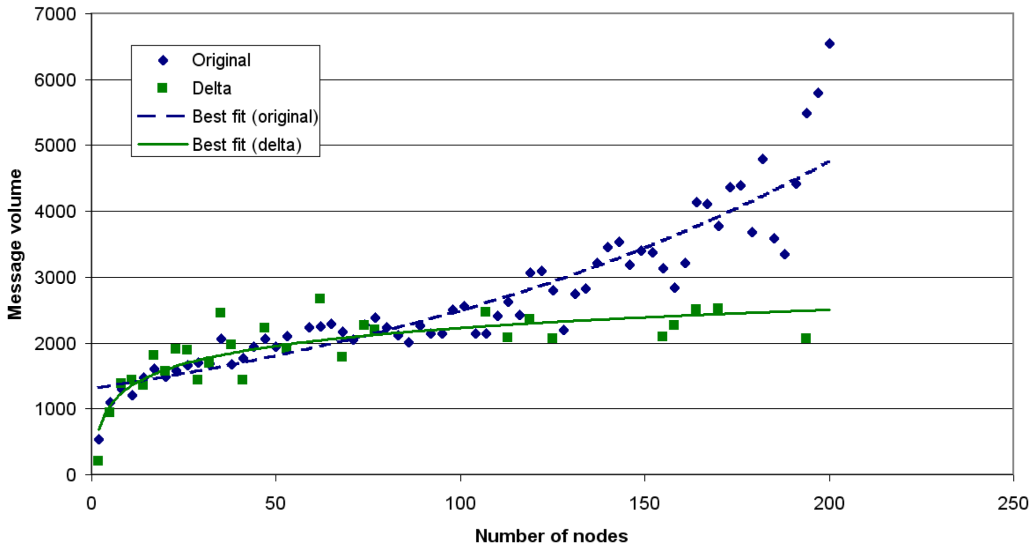


Figure 5.18: Comparison of delta optimised and original solutions for rings of size 20

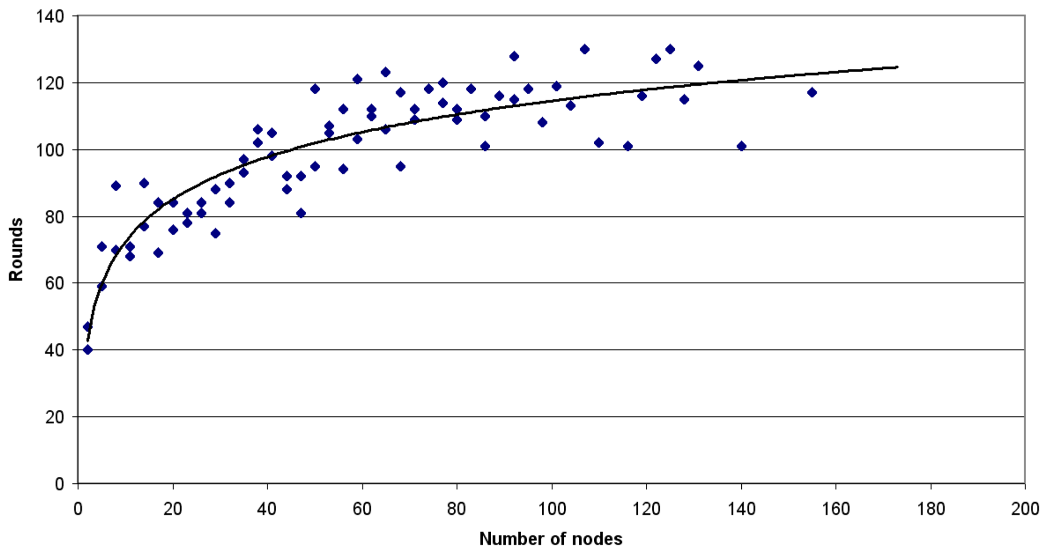


Figure 5.19: The number of rounds taken using the delta optimisation where $c = 20$

5.4.2 Configuration-Directed Gossip

The performance of the algorithm in terms of the number of rounds may be improved by modifying the pattern of message exchange. For example, the state held by each node contains the names of the nodes hosting components involved in the configuration, and this might be used to choose message recipients more efficiently. Intuitively, if a node decides to modify the state, it is most important to inform the other nodes mentioned in the state, rather than nodes who are not involved. For instance, if a node modifies the state to satisfy a requirement $(\mathbf{req}, (c, n), i)$ then the named node n is most "interested" in an update, perhaps because the satisfaction allows it to instantiate or resume execution of c (after adapting to a lost provision).

Of course, all nodes must be informed eventually. Firstly, this is to ensure all possible solutions are considered. Secondly, it remains necessary for the entire network to agree on a solution otherwise (with a new definition of convergence restricted to named nodes) different groups of nodes may converge on different solutions. If these configurations are instantiated, the user may find that two systems, competing either in computational or environmental resources, have emerged.

The benefit of this optimisation remains to be tested since changing the message exchange pattern can have a significant effect on the performance.

5.4.3 Variations On Gossip

There are several opportunities to improve the performance of the algorithm by using different variants of gossip. As mentioned above, rumour mongering can be used in place of decentralised two-phase commit, if a small probability of error is permissible. Also, the algorithm uses a push policy for propagating updates, which might be improved upon by using a push-pull policy whereby updates pass in both directions. Changing the pattern of message exchange from a uniform random choice across all nodes to a configuration-directed pattern as above, or to another general pattern such as round robin, may also provide a benefit.

Each of these variations will have a different effect on performance, which must be tested in particular domains. However, the approach is sufficiently modular that these variants can be used without changing the overall scheme.

5.5 Summary

This chapter has presented a gossip-based technique for decentralised assembly and adaptation whereby a network of peer nodes co-operate to derive and ultimately agree upon a configuration meeting the functional and structural constraints. Agreement on a solution occurs within $O(s \log n)$ synchronous rounds (where s is the size of the configuration and n the size of the network) if extensive backtracking (resulting from structural constraints) does not occur. Gossip is robust to failure of nodes and can adequately cope with 85% message loss. This work addresses requirement 5 (distribution). The next chapter describes the mechanisms used, once a decision has been made to switch from one configuration to another, to guarantee the safety of the application and to preserve application state.

Chapter 6

Runtime Adaptation

AUTOMATIC assembly as described above permits the system not only to synthesise an initial software configuration but also to derive a new target configuration in order to adapt to changing circumstances. While in this work we do not circumscribe which events are to be regarded as a cause for adaptation, we can suggest four categories which are likely to be relevant in most domains: environmental change, evolution of requirements, dynamic availability and monitoring of non-functional properties. We describe these four circumstances, and then consider the detail of how a reconfiguration is derived. We discuss how primitive transformations, such as connection and disconnection, are selected to effect the switch from the old configuration to a new one, while simultaneously being careful not to disrupt or endanger normal component execution. In addition, we show how the state (both in the sense of status and of stored data) of a failed component can be preserved for use in its replacement, where applicable.

Finally, having completed the description of the principles of the approach, we describe the prototype implementation in the last part of this chapter.

6.1 Causes For Adaptation

6.1.1 Change Of Requirements

Since our approach is declarative—goals are transformed into functional requirements and eventually to components—it supports adaptation in response to changing user requirements. An explicit goal statement is often much shorter and easier to change than its counterpart in approaches which lack this declarative feature. For example, in Rainbow [Che08], a complete new set of repair strategies would have to be written. Once the functional requirements have been modified, reconfiguration can be invoked to generate a new configuration (assuming of course that the components relevant to the new requirements exist).

6.1.2 Environmental Change

This broad category of events can affect the system in a variety of ways, leading to different mitigating efforts. For example, components may make assumptions about the physical world or about the availability of certain hardware facilities. Such assumptions are easily invalidated, having the effect of making that component unavailable (see dynamic availability). The breaking of assumptions may be detected explicitly by the components, so that they fail gracefully, or the system may simply detect that the component has failed for an unknown reason.

The environment can also have an impact, for similar reasons, on the non-functional properties of a component. For example, high network congestion can reduce the claimed performance of a component. Again, the relevant environmental property or else the consequent NF properties of the component may be monitored (see NF monitoring).

6.1.3 Dynamic Availability

Like all software, components are prone to implementation bugs which may prevent their use under certain conditions. They also make assumptions about their operating environment that may be broken when deployed. These situations can cause partial or total failure of the component. Given some mechanism for detecting component failure, such as explicit exception handling or polling to check the component is “alive”, such events may be used to mark a component as unavailable, and to invoke re-assembly. The failed component will then not be selected in the re-assembly process.

6.1.4 NF Monitoring

Incorrect annotation or unexpected operating conditions can cause the non-functional properties of a component to diverge significantly from the annotations used during assembly. For example a component might claim to be reliable, and yet repeatedly fail in a particular deployment environment. In such a scenario, the configuration no longer meets the expectations that the user expressed through NF preferences and so re-assembly should be invoked in order to continue meeting those expectations. As with component failure, the components must be monitored, either internally or externally, to determine when (and to what degree) the component is no longer behaving as its NF annotation suggests. When this occurs, the annotation for the particular monitored property should be updated with the correct value and re-assembly should be invoked to choose a component with higher utility (if there is one).

Unfortunately, NF monitoring is complicated by the general nature of NF annotations permitted by the approach, and by the need to preserve component encapsulation. We cannot hope to provide a standard monitor for all conceivable NF properties, and so there must be a way for designers to create monitors for the properties which interest them (if those properties are indeed subject to fluctuation). While this increases the complexity of the approach from the viewpoint of system designers, it does provide a high level of flexibility.

In reality there may be a wider range of reasons to initiate adaptation, but, as in these four cases, the principle remains that a reconfiguration should occur whenever the inputs to the assembly process change (significantly). These inputs include the set of available components (so a newly-available component may also cause an adaptation), the functional requirements (handling a change of goal), the structural constraints, the non-functional annotations, and the designer's preferences amongst the NF properties. The subsequent sections show how, once initiated, a reconfiguration is computed and applied.

6.2 Centralised Adaptation

In the centralised algorithm, new configurations are derived by running the assembly procedure with updated availability, functional requirements or non-functional annotations. The new configuration is then compared against the existing configuration to derive the changes which must be performed.

6.3 Distributed Adaptation

In the distributed case, adaptation is slightly more complex. Firstly, if the adaptation is a result of changing functional requirements, the assumption is that every node receives the updated changes in the same manner in which the initial requirements were received.

In the case of a change detected locally, such as a failed component or diverging NF properties, only a single node need observe the change. This node must force the network to choose a new solution by issuing a death certificate[1] for the failed configuration and selecting an ancestor state exactly as if the node was backtracking after checking the structural constraints (rule 3(a)). Additionally, the node must update the availability or NF properties of the offending component. Then, as with backtracking, the network will avoid the failed solution to converge on a different one.

6.4 Deriving The Changes

The manipulations which are specified explicitly in other approaches, such as component deletion or replacement, connection and disconnection, are subsumed by our approach, which is to regard component interconnection as implicit, and the configuration as a set of components.

Once a new configuration has been generated by running the assembly procedure with updated availability or non-functional annotations, the specific actions to be performed can be easily derived. The new components to be instantiated (the "delta") are those in the difference between the old and the new configurations. Likewise the old components to be destroyed are those in the commutated difference:

$$createComponents = \mathit{newConfig} \setminus \mathit{oldConfig}$$

$$destroyComponents = \mathit{oldConfig} \setminus \mathit{newConfig}$$

All connections to or from a destroyed component must be removed, and new connections to or from new components are created using the same logic applied for the initial construction (whereby a requirement is satisfied by any provision).

[1]The metaphor takes on particular aptness here.

6.5 Safe Update

Ideally, the components which are present in both the old and new configurations should be permitted to continue execution, rather than disruptively pausing the entire application for the duration of reconfiguration (or worse, discarding the entire old configuration and re-instantiating those components). Pausing the application in this manner may incur a cost in terms of the start-up time of components, or in some domains it may not be safe. For example, an unmanned airborne vehicle cannot safely stop its flight controller while airborne. The components to be preserved are:

$$preservedComponents = oldConfig \cap newConfig$$

Equally, it is not safe to allow every preserved component to continue in ignorance of the changes. A preserved component may be connected to another component which is about to be replaced. There are several reasons why replacing a dependency such as this may be unsafe, including:

- The preserved component may attempt to interact with its dependency at a moment when it is not bound (similar to a dangling pointer). The transaction will fail.
- The preserved component may already be interacting with the dependency at the moment that it is replaced. Again, the transaction will fail.
- The preserved component may have interacted with its dependency, and may have the intention to interact with it again to complete the current computation. Even if the replacement is completed before the preserved component needs to use the dependency again, this may not be safe if the transaction relies on the requirement being satisfied by the same component throughout the computation. A very simple example would be any component

which provides a protocol with open and close operations like a file or socket. If the file component is opened, then replaced, then the new component will not be in the correct state to perform a close operation[2].

Of course, there are additional reasons for having some means to warn a component about its imminent destruction:

- The component may be in the middle of an internal computation.
- The component may need to perform some operations to terminate safely, such as closing files and releasing resources.
- The component may wish to preserve its internal state.
- The component may continue trying to use bindings that have been disconnected.

To combat these issues it is necessary to employ some protocol either to pause some of the preserved components, or to find a safe moment for adaptation, so that the changes can be applied without jeopardising the safety of execution.

6.5.1 Tranquility

This problem can be addressed (in the centralised case) by applying one of the schemes described in Section 2.5. We use the tranquility protocol (Section 2.5.6) since this causes a minimal amount of disruption to ongoing application execution. This section describes the principles of component replacement using tranquility, while the implementation details are given in Chapter 6.7.

First consider the replacement of a single component which has a single provision, bound to a single requirement of another component (one which is to be preserved).

[2]Though this may be mitigated if the new component adopts the preserved state of the deleted component.

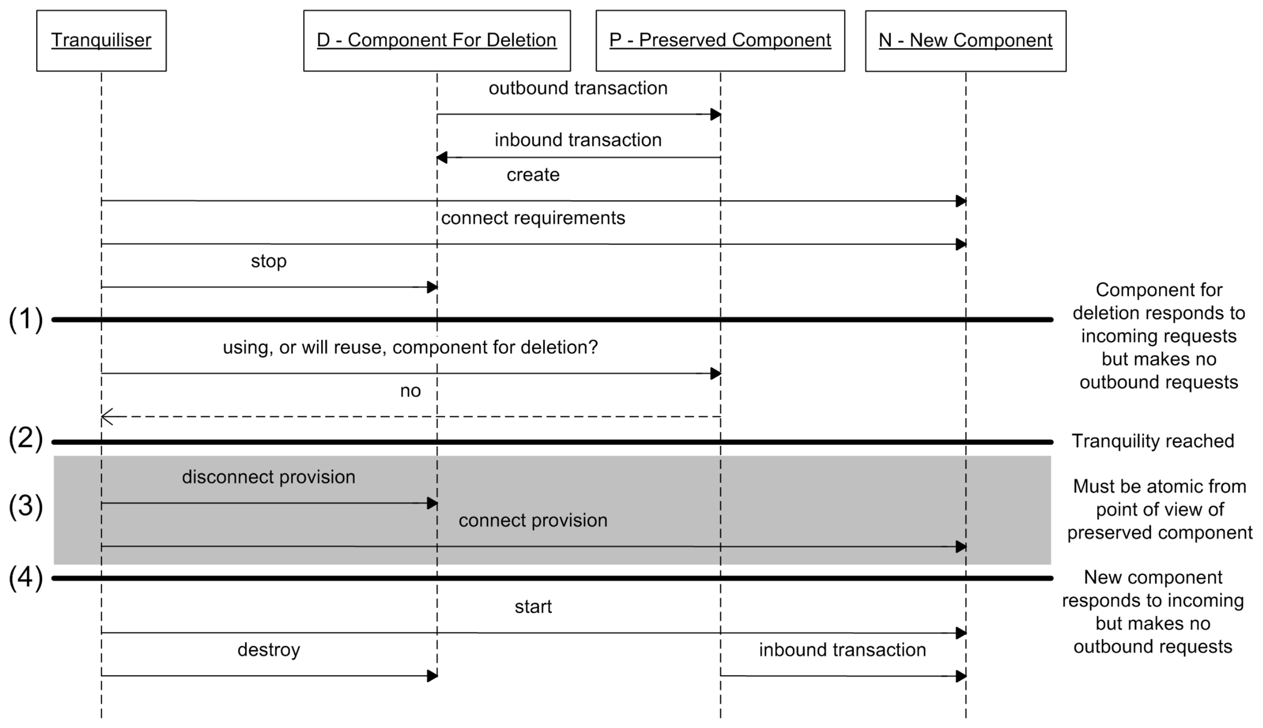

Tranquiliser
D - Component For Deletion
P - Preserved Component
N - New Component
outbound transaction
inbound transaction
create
connect requirements
stop
(1)
Component for deletion responds to incoming requests but makes no outbound requests
using, or will reuse, component for deletion?
no
(2)
Tranquility reached
disconnect provision
(3)
connect provision
Must be atomic from point of view of preserved component
(4)
New component responds to incoming but makes no outbound requests
start
destroy
inbound transaction

Figure 6.1: Tranquility protocol (centralised)

The sequence chart in Figure 6.1 displays the protocol for this case. The first two messages of the diagram show normal execution, in which the component to be deleted (D) receives requests from its dependant (P) and makes requests to other components. At some point the "tranquiliser" (implemented within the configuration assembler; not as part of the application) decides to replace D with a new component N. Since it is desirable to minimise disruption, N is instantiated and its requirements bound (though they will not be used) before interrupting normal execution. Then, the tranquiliser instructs D to stop making outbound requests (point 1), though it may continue servicing inbound requests from P. Then, the tranquiliser must wait for the system to enter the tranquil state in which D is not currently involved in any transaction and will not be used in an ongoing transaction if it has already participated. Recall from Section 2.5.6 that it is possible (though unlikely) that tranquility will never be reached, requiring the system to fall back to using the more disruptive quiescence protocol. To detect the tranquil state, the tranquiliser repeatedly queries the dependant P. When P reports that it is not using and will not reuse D, it is a safe moment to switch the binding of P's requirement to the new component N (point 2).

Notice that if there is an instant (point 3) in which the old binding to D has been removed and the new binding to N has not been created, then it is possible for P (which has not been made passive in any way) to initiate a request upon an unbound requirement. Hence, this operation must appear to be atomic from the point of view of P. This can be achieved by momentary blocking of the connection (as in Section 2.5.5), but in our case we exploit the fact that a binding is implemented as an ordinary Java reference which can be overwritten in an atomic step.

At point 4, the new configuration is fully connected but the new component N has not been instructed to begin execution. Before this point, symmetrically with the demise of D, it may process incoming requests, but not make any outbound requests. The tranquiliser informs N that it can begin by sending it a start message. Normal execution of the new configuration continues while the tranquiliser may perform any clean up operations required to destroy D at its leisure.

Now consider the case when a component like D has multiple provisions, or equivalently, multiple components are bound to its provision. In these cases the protocol repeats the section between points 1 and 4 for each binding. While this means that there is a period of time in which some components are bound to the old provision and some are bound to the new, this is in principle safe since components should not be aware of their context including which provisions their "peers" are bound to.

It is also possible to apply tranquility in the distributed case, where extra messages are required to co-ordinate the various nodes. In this case, each node has its own tranquiliser. We do not provide each component with its own tranquiliser for reasons of efficiency. In Figure 6.2, the tranquiliser TN local to the new component N is responsible for its creation and connection, and the tranquiliser TD local to D is responsible for stopping D. At this point an extra "ready" message is sent from the two mentioned tranquilisers to the one local to the preserved component (TP).

For this purpose it is necessary that TN knows TP (and vice versa) and that TD knows TP. Notice, however, that there is no interaction between TN and TD. The names of the nodes can be found from the gossip state[3].

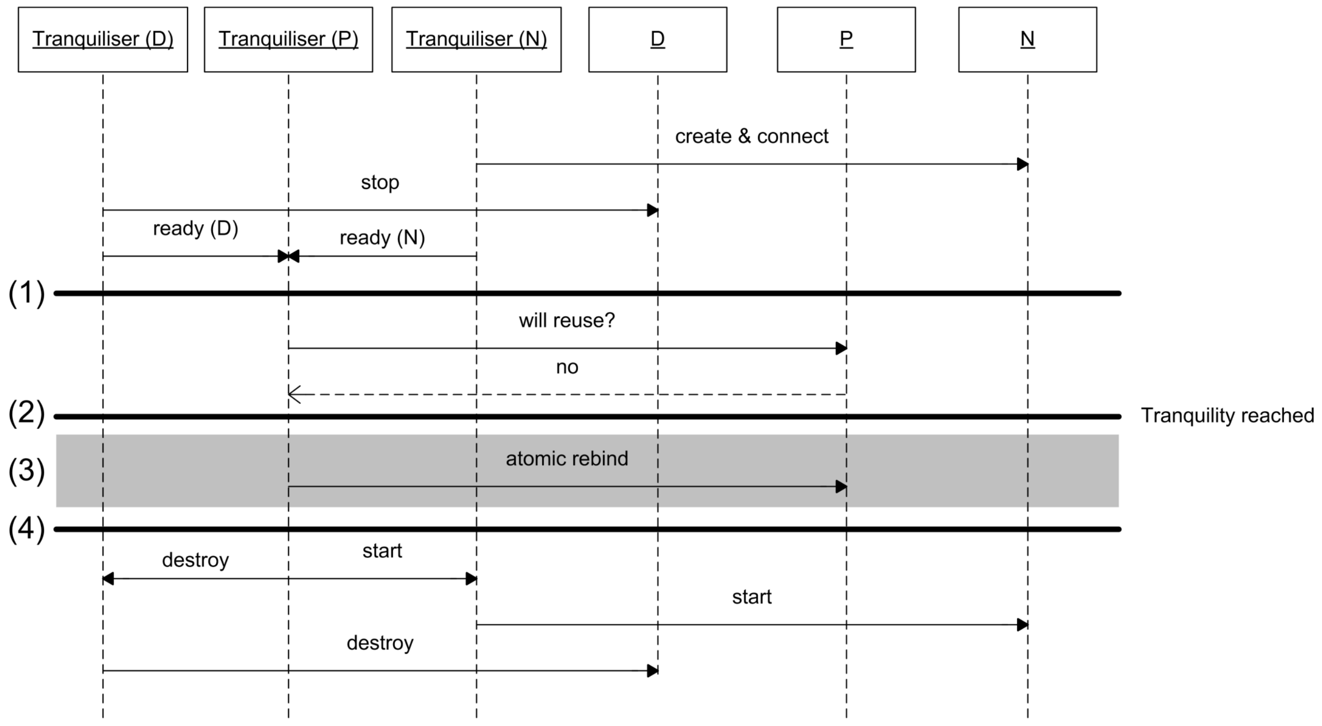


Figure 6.2: Tranquility protocol (distributed)

Once the “ready” messages have been received by TP, it can query P and perform the atomic rebinding as above. To complete the process, TP informs TD that it may destroy D, and informs TN that it may start N.

6.5.2 Multiple Replacements

Tranquility as originally described only accounts for the replacement of a single component in a centralised setting. However, in general, a new configuration will be comprised of several components which are entirely new and several which replace existing components, in addition to those which are retained from the previous configuration. This demands that we consider what constrains the manner in which multiple replacements are performed.

[3]This information is in any case a prerequisite for creating the bindings in the original configuration.

The key observation is that performing the entire process for each component separately will cause unnecessary disruption since the time between the first rebinding and the last must now include component start-up times. This approach would also mean that the instantiation of new components would have to follow the (structural) dependencies since there cannot be an instant in which a requirement is unbound. Components which are depended upon would have to be instantiated before those which depend upon them. Cyclic dependencies further complicate the situation. Our approach simplifies the situation by processing all the new components together, in four phases as described below.

Firstly, all new components (and those which will replace old components) are instantiated, but are not started. They may perform internal computations, but it is assumed they do not make outbound requests. All the requirements of these new components are bound to other new components, or to old components. If there is a new component which provides the same interface as an old one, then the new component is chosen for binding since the old component may be replaced (thus binding to it would cause an unnecessary rebinding). At this point, the old configuration remains in operation, and a partial new configuration, which is disconnected from the original, is partially operating. Notice that from this point, and until the reconfiguration is complete, the actual configuration may violate the structural constraints.

In the second phase, the new configuration is patched into the old one by iterating over all the bindings to components to be deleted, and performing steps 1 to 4 of the tranquility protocol to transfer each binding to the new components. This leads to a sequence of configurations where the new sections of the configuration are partly connected to the preserved configuration. This is safe since each rebinding is an atomic step, and so at every stage of the sequence, a binding is either to an old component or to a new one, and so requests can always be serviced (new components can service inbound requests). There is no observable point at which a requirement is unbound.

In the third phase, each of the new components is sent the start message which permits them to make outbound requests. It is the absence (or blocking) of consequent transactions which allows components to be started in any order. If transaction dependencies were handled at this level, then the components would have to be started in the reverse order of the chain of dependencies, so that requests always go to a component which has already started. If indeed there are any consequent calls, they must be blocked by components which have not yet been instructed to start. This means that, in the worst case, a transaction could be blocked from the first rebinding until the last start message. However, rebinding and the sending of start messages are not computationally expensive operations, and so the delay should be minimal. At this point, the new configuration has been completed and once again satisfies the structural constraints.

The final step is to terminate all the old disconnected components, concurrently with the normal execution of the new configuration which has been assembled.

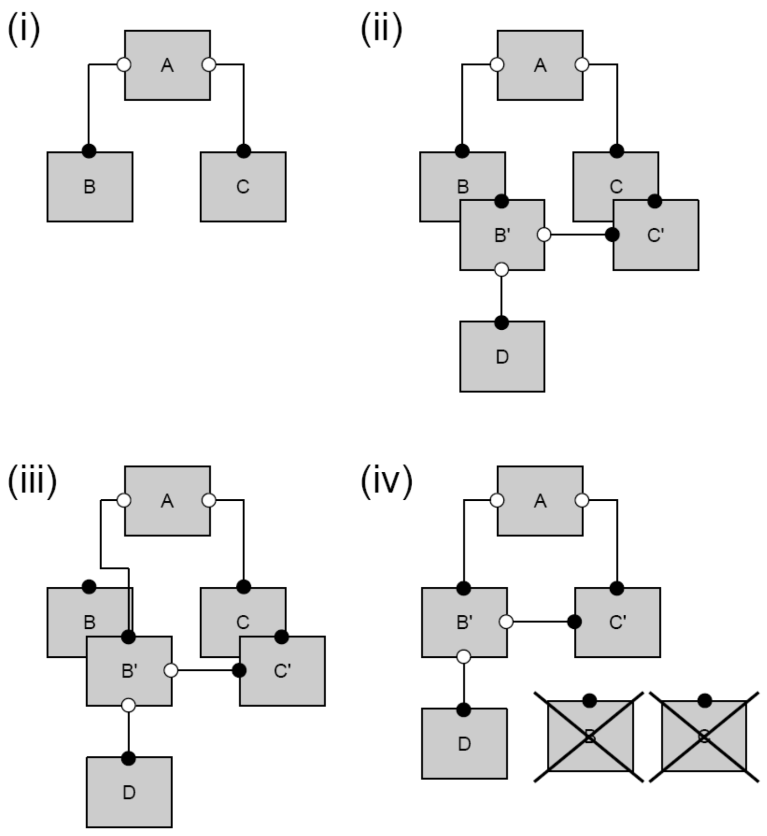


Figure 6.3: Performing multiple replacements safely

Figure 6.3 shows an example in which two components, B and C, must be replaced by B' and C'. Figure 6.3 (i) shows the initial configuration. In step (ii), all the new components are instantiated, and their requirements are bound. Notice that B' introduces two new bindings not found in the initial configuration, one of which is satisfied by a completely new component D.

In step (iii) the tranquility protocol is performed for the binding of the preserved component A to B'. B' may service incoming requests from A but may not make any outbound requests to C' or D. In step (iv), the binding between A and C is moved to C', completing the new configuration. It is now possible to discard the old components B and C, and to instruct B', C' and D that they are now permitted to make outbound requests.

6.6 State Preservation

To increase the likelihood that a reconfiguration causes a minimum of disruption, we provide a means for deleted components to preserve their internal state[4], and to have that state restored in new components. To see the motivation for this, consider a situation in which one component has interacted with another several times, and the manner of these interactions is determined by the state of the components, which may be a simple status (such as "done the first step") or a complex data structure. If one of these components is replaced, even in a safe manner as above, the new component will be in its initial state which does not correspond to the state in the preserved component. It is likely that the ongoing interaction between them will have to be restarted to reconstruct the internal state. For example, in the robotics domain, there may be a component which constructs a map of the environment from one type of sensor (say infra-red), which is then replaced by

[4]This is a different sense of "state" from that used in the discussion of gossip. Here we refer to the component's application state, while the state of gossip is global configuration information stored in the configuration assembler of each node.

another component which uses a different source of data (such as a webcam). Clearly it is desirable for the map data from the first component to be retained for use in the second component.

To avoid disrupting the system in this way, the old state can be passed to the new component, in the hope that the new component will adopt this as its own state. Unfortunately there is no guarantee that the new component will pay attention to the old state. It may not be a meaningful data structure to the new component (indeed one expects different implementations to have different internal representations), and the system cannot force the new component to adopt the state without either (i) breaking the black box principle by peering into its implementation details, or (ii) losing generality and flexibility by handling a restricted class of state information which can be understood by most components.

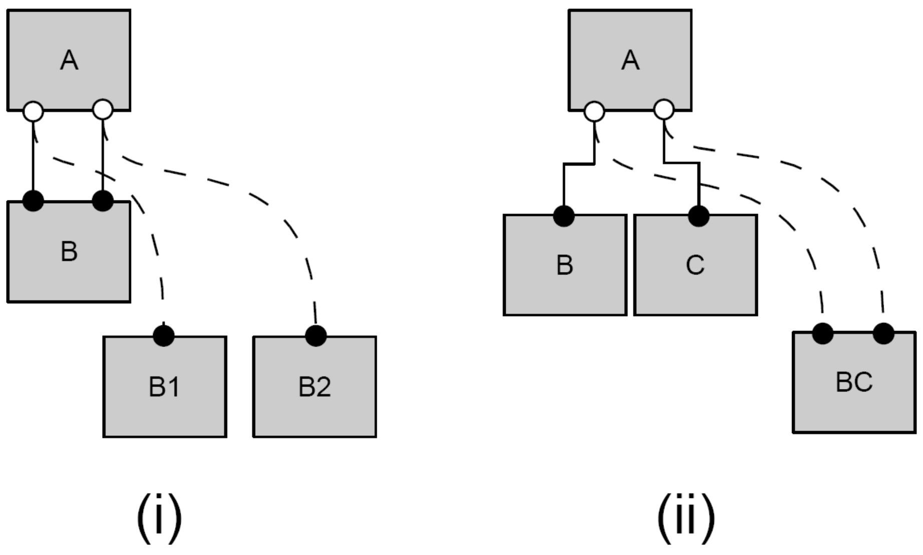


Figure 6.4: New components may split (or unify) the provisions of an old component amongst themselves

Hence, the system requests from each component that will be deleted a reference to an arbitrary data structure, and when instantiating a new component, the system provides this reference. However, there is an additional complexity in that in general there is no direct relation between an old component and the component replacing it, complicating the decision of which new component(s) to pass the old state to. This arises because a single component may implement many interfaces, and so any new component may replace another with respect to only one of its provided interfaces. For example, in Figure 6.4 (i), the component B implements two interfaces and these are implemented by different components B1 and B2 in the new configuration. In (ii) the interfaces provided by components B and C are combined in a single component BC.

Rather than make the (likely futile) demand that components maintain and isolate different states for each of their provisions, the old state is passed to any component which implements an interface of the old component. This means that in situation (i), both B1 and B2 will receive the state of B; and in situation (ii), BC will receive both the state of B and the state of C.

6.7 Implementation

Although in the above chapters the abstract behaviour and performance characteristics of our approach have been described, with the intent of making the efficacy and feasibility thereof manifest, there remains to be discussed some particulars of our prototype implementation.

6.7.1 Dynamic Backbone

In order to implement our assembly and adaptation techniques, it was necessary to use a framework that permits the definition of components and the instantiation and manipulation of configurations (involving component instantiation and wiring). For this purpose we made use of the Backbone framework [MKM06] by Andrew McVeigh and extended it with the assembly algorithms and the tranquility and state preservation protocols.

The original purpose of Backbone was to instantiate configurations designed in a graphical modelling tool called Jumble. A diagram created in this tool is essentially a static description of a configuration, save for one construct called a factory, which permits a limited sort of runtime change and constructs which describe the evolution of the design (again, statically). These constructs have been discussed in detail in Section 2.2.1.

Given the static nature of these constructs, our strategy has been to avoid using any specific Backbone features (which nevertheless would only provide a means, and not absolve us from having to make adaptive decisions). This ensures that the approach remains generally applicable across a wide range of architectural frameworks. The sole necessity is that components are explicitly marked with their provided and required interfaces.

Backbone is implemented in Java and requires that components are also implemented (at least nominally) in Java. Figure 6.5 is a class diagram indicating the major classes involved in the extended implementation along with their significant methods and fields.

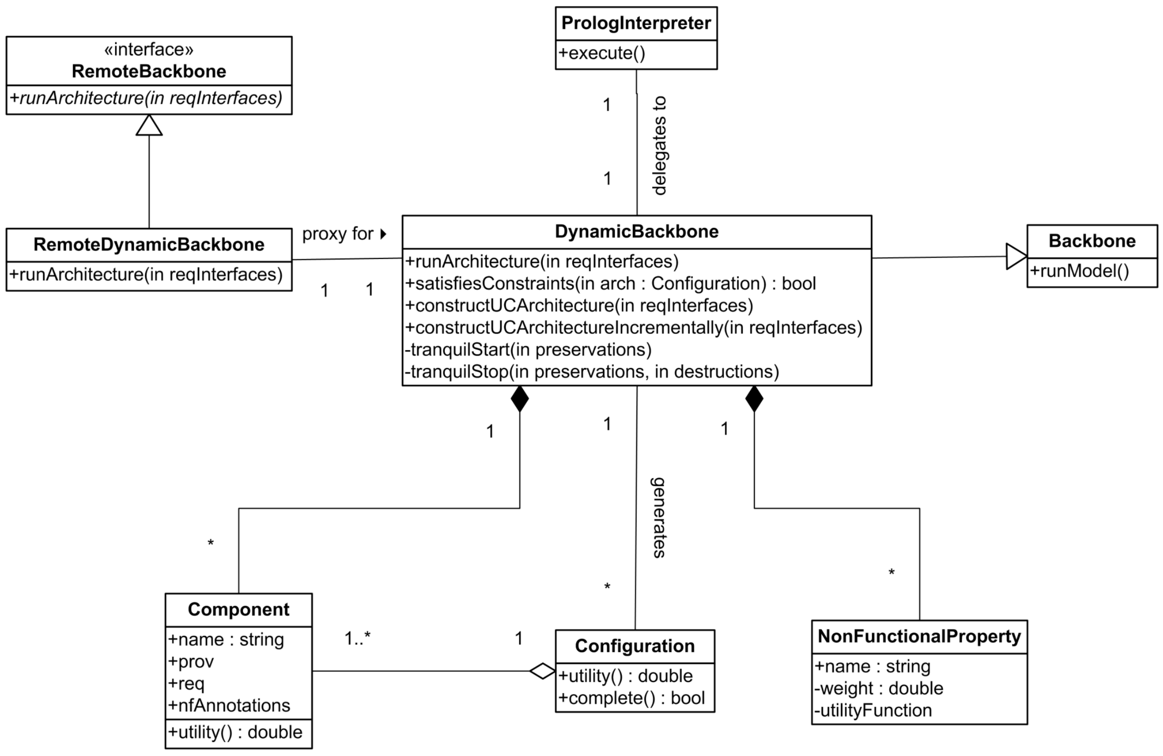


Figure 6.5: Dynamic Backbone

The most important class is DynamicBackbone, which extends the original Backbone class. The Backbone class has a method runModel() which instantiates, in the local Java Virtual Machine, the configuration described in the architectural model passed as a parameter. In its original mode of operation, Backbone would read the configuration from a set of files written in Backbone's ADL, construct the model, and then call runModel(). DynamicBackbone circumnavigates the Backbone ADL and constructs models directly at runtime, thereafter calling runModel(). Note that DynamicBackbone only supports the centralised algorithm at present, though it should be a simple matter to add the distributed version.

RemoteDynamicBackbone is a remote proxy for the DynamicBackbone, implementing the RMI (remote method invocation) interface RemoteBackbone, thus allowing remote access.

Component, Configuration and NonFunctionalProperty are data structure classes describing the relevant entities (thus, collectively they can form architectural models). DynamicBackbone has a collection of Component instances[5] that form its component repository and a collection of NonFunctionalProperty instances that describe all the utility functions and weights for the application. DynamicBackbone is responsible for generating Configuration instances (composed of Component instances) which are then instantiated by Backbone.

The main entry point for assembling configurations is the method runArchitecture() which takes a set of required interfaces (String objects) as a parameter. This method calls either constructUCArchitecture() or constructUCArchitectureIncrementally(), performs the tranquility protocol and ultimately calls runModel(). constructUCArchitecture() performs the assembly procedure using aggregate NF selection, whereas constructUCArchitectureIncrementally() does the same using incremental selection. Both these methods call satisfiesConstraints() which checks that the candidate configuration meets the structural constraints by

[5]Note that a Component object merely describes a component *type*, and is unrelated to instances of real, operational components.

making appropriate calls to the `PrologInterpreter`. We use GNU Prolog for Java[6], which is implemented completely in Java, but this can easily be substituted with alternatives.

The tranquility and state preservation protocols are handled by the `tranquilStart` and `tranquilStop` methods. `tranquilStart` simply instructs new components to begin execution and injects preserved state. `tranquilStop` waits for a safe point to apply changes before stopping the components to be deleted and extracting state to be preserved from them. This is achieved—for each component d to be deleted—by:

- waiting for a point when d is not engaged in an external or internal transaction (by querying d and intercepting incoming calls using a `java.lang.reflect.InvocationHandler`),
- instructing d to stop normal execution (it may still respond to incoming transactions), and
- waiting for each component dependent upon d to assert that it will not reuse d in the current transaction.

At this point the binding can be changed, and d destroyed.

6.7.2 Domain Components

Components used with `Backbone` must be implemented in Java (or at least have a Java wrapper[7]), and must follow a number of conventions. These conventions were kept to a minimum in order to reduce the burden of creating Backbone components. The conventions are:

- When a component C provides an interface I, it must *implement* I according to the normal Java rules. This means that a component can only implement one

[6]`http://gnuprologjava.sourceforge.net`
[7]This is the case for the Koala component of Section 7.1.

copy of an interface, though we do not believe this to be a serious practical problem (in other words the notion of a port for provisions is lost).

- When a component C requires an interface I, it must have a public non-final field with the type `I` and marked with a `req` annotation including the name of the port. For example `@req("portName")` marks a requirement named `portName`. The `req` annotation type is in the Java package `com.hopstepjump.backbone.api`, meaning that an appropriate `import` statement is also needed.

- If a component wishes to have the safety of tranquility, it must additionally implement the `TranquilComponent` interface. This requires suitable implementations of the `startComponent`, `stopComponent`, `transaction-InProgress` and `getReusedComponents` methods. This necessitates importing `com.hopstepjump.backbone.api.TranquilComponent`.

- Likewise if a component wishes to allow its internal state to be saved and restored, the `StatePreservingComponent` interface, comprising `saveComponentState` and `restoreComponentState` methods, must be implemented. The first method must return an `Object` (reference) and the second must takes an `Object` as a parameter. `com.hopstepjump.backbone.api.StatePreserving-Component` must be imported.

Figure 6.6 shows the source code for the ObstacleAvoider component of the Koala case study. The ObstacleAvoider provides the MotionSource interface, and requires a MotionController and KoalaSensors via ports named `controller` and `sensors` respectively. The ObstacleAvoider does not strictly need tranquility, but it uses the `startComponent` method as a notification that its requirements have been bound and hence it can call the MotionController.

Components are naturally permitted to start their own threads, and so the `startComponent` and `stopComponent` methods give a suitable place for starting threads and cleaning up after them. The `transactionInProgress` method should return true if any thread is involved in a lengthy computation or if it is interacting with a dependency. For transactions which involve multiple calls to dependencies, the component must ensure that the dependencies are identified in the `getReusedComponents` method which prevents `DynamicBackbone` from destroying any requirements which are in use.

6.8 Summary

In summary, the major part of this chapter has described the conditions under which an adaptation is to be performed, and how a reconfiguration is applied safely. An adaptation can be invoked by a change of requirements (which may be a result of a new goal), by an environmental change, by components ceasing to be available (after failures), and by monitored non-functional properties diverging from their ideal values. These events cause the assembly process (centralised or decentralised) to resume and compute a delta consisting of the changes to be made to the existing configuration to arrive at the new one. The safety of the changes (with respect to preserved components) is ensured by using an extended version of tranquility that handles multiple replacements and distribution. The preservation of application state in replaced components is also catered for with a scheme for extracting and injecting state information. The work in this chapter addresses requirement 4 (safety) given in the introduction.

The final part of the chapter discussed the implementation of the techniques described in this and the preceding chapters. The next chapter applies the centralised and decentralised techniques to two case studies and evaluates the work.

```java
package koala.motion;

import koala.KoalaSensors;
import koala.KoalaSensorReading;
import koala.KoalaVector;
import java.util.Vector;
import com.hopstepjump.backbone.api.req;
import com.hopstepjump.backbone.api.TranquilComponent;

public class ObstacleAvoider implements MotionSource, TranquilComponent
{
  @req("controller") public MotionController controller;
  @req("sensors") public KoalaSensors sensors;
  private final int obstacleThreshold = 150;

  public ObstacleAvoider()
  {
  }

  public void startComponent()
  {
    System.out.println("ObstacleAvoider:␣Registering␣with␣motion␣controller.");
    controller.registerMotionSource(this, 50);
  }

  public void stopComponent() {}

  public boolean transactionInProgress()
  {
    return false;
  }

  public Vector<Object> getReusedComponents()
  {
    return null;
  }

  public KoalaVector getDirection()
  {
    //implementation omitted
  }
}
```

Figure 6.6: Source code for the ObstacleAvoider component

Chapter 7

Evaluation

AFTER having discussed the technical and theoretical aspects of our declarative approach for architectural adaptation, in both the centralised and distributed forms, it remains necessary to provide a qualitative evaluation of the utility and efficacy of the approach with respect to both our requirements and to potential application areas, complementing the quantitative results given in prior chapters. To that end, this chapter presents two case studies (drawn from a larger set of experiments performed in the course of the work), and then a discussion of how each objective was satisfied, particularly in relation to the case studies. The first case study is one in the robotics domain in which mobile robots are given a goal to achieve in the physical environment, and the second case study is for satellite tracking and was originally proposed elsewhere [BHRE07].

Finally, a number of limitations are discussed. Some of these relate to observations drawn from the case studies, or otherwise follow from our assumptions (Section 4.1).

7.1 Robotics

7.1.1 Scenario

Systems with some robotic element—unmanned vehicles for example—are a natural application domain of autonomous systems, since they are especially faced with unpredictable and uncertain environments.

Suppose there is a fixed robotic arm and two mobile robots. In our demonstrations we have used the Koala robot[1], so we refer to the mobile robots as Koalas[2]. These robots are equipped with a standard set of components representing a wide range of functionality. The user provides a goal, from which a reactive plan is generated (following the three-layer model), which results in a set of functional requirements.

Figure 7.1 shows the complete set of components arranged into a dependency graph. Some of these components can only be used on hosts providing the relevant hardware: the Koala component provides access to the hardware of the Koala robot while the KatanaArm component does the same for the robot arm. The Koala is capable of motion in a two-dimensional plane (using the KoalaMotors interface) and sensing the presence of obstacles using reflective infra-red sensors with a range of approximately 20cm (using the KoalaSensors interface). The robot arm is capable of moving such that its gripper can be positioned at a three-dimensional point lying within a sphere centred on the "shoulder" of the arm (though there are several physical limitations such as its not being able to grip too close to, or underneath, the shoulder). The arm is also capable of opening and closing the gripper, and has a set of infra-red and pressure sensors, which can be used to detect gripped objects. The arm functionality is accessed through the RobotArm interface. Both kinds of robot may additionally make use of a camera to enhance their sensing capability. Most often this is achieved by using the Webcam component which allows images

[1]`http://www.k-team.com`

[2]Videos of our demonstrations can be found, at the time of writing, at `http://www.doc.ic.ac.uk/~das05`

to be grabbed from the camera.

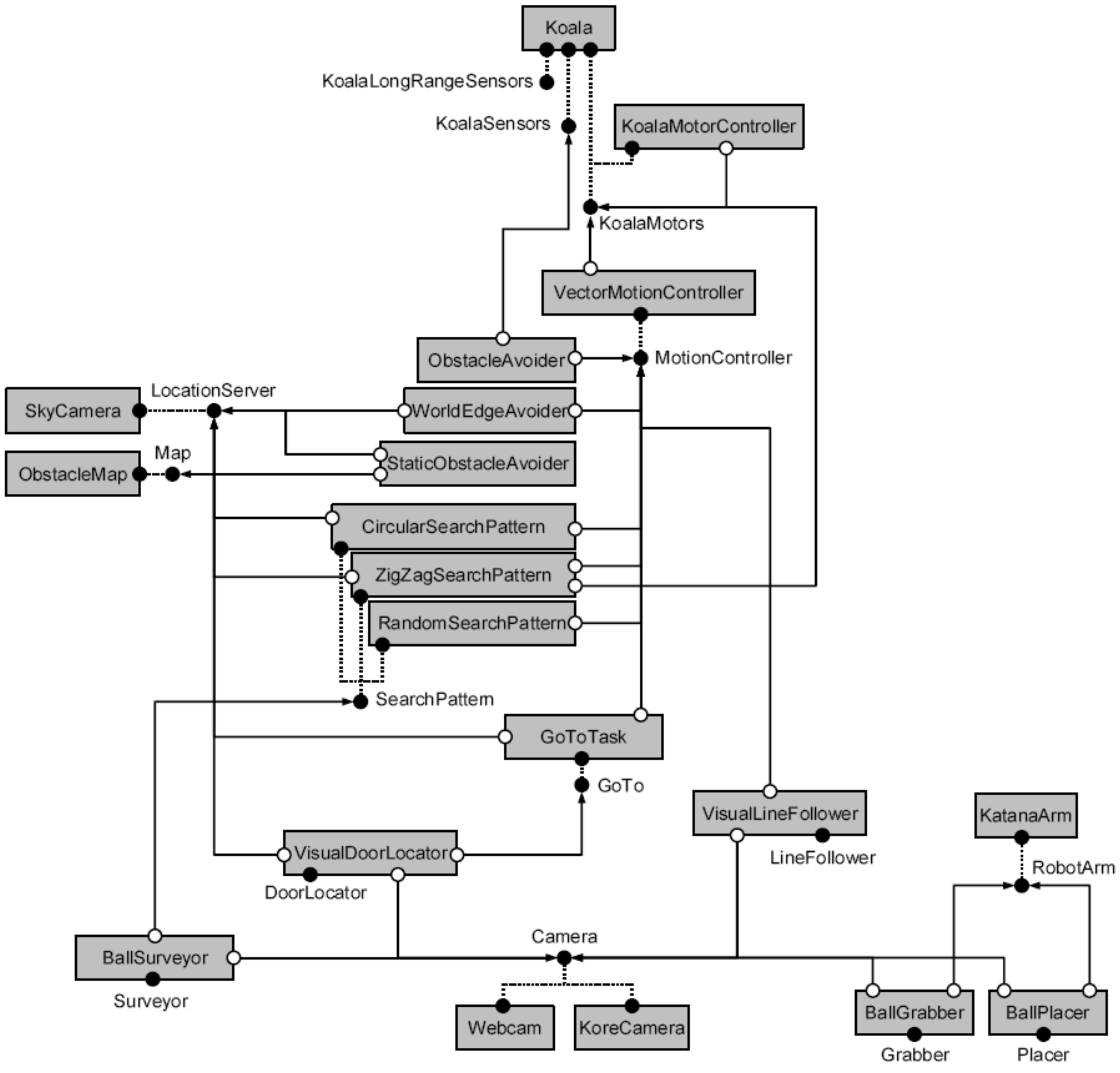


Figure 7.1: Robotics components

From these fundamental behaviours, various composite behaviours can be formed. Several of the components are concerned with moving the Koala according to different rules, such as a circular pattern useful for surveying an area. The ObstacleAvoider interprets the infra-red sensors to adjust the motion of the Koala according to the proximity of obstacles. These motion-affecting behaviours are compositional through the use of the VectorMotionController, which computes a weighted sum of the desired trajectories of each subsidiary component. This sum is used to set the speeds of the wheels so that the actual motion reflects all the behaviours.

Several of the motion-affecting components require the (absolute) position or orientation (heading) of the Koala which can be found through the LocationServer interface. In our implementation, locations are detected using external infrastructure that uses cameras to detect locations, though equally a GPS device and a digital compass might be used.

The remaining components are largely concerned with high-level non-compositional behaviours, and implement the abstract functional requirements used at the planning level. For example, the GoToTask moves the Koala to a specified position (hence requiring the LocationServer), the BallSurveyor uses a SearchPattern to control the motion while using a Camera to detect coloured balls in the arena, and the BallGrabber uses a Camera to detect coloured balls and the KatanaArm to pick them up.

The possible functional requirements in this system are then subsets of {GoTo, Surveyor, DoorLocator, LineFollower, Grabber, Placer, Camera}. Goals based on this set can include such examples as moving to a series of specified points and taking photographs (which might be used to search for explosives in a military context), driving through mazes, and carrying objects from one location to another.

To illustrate our approach, we choose one particular goal (and suppose the existence of a plan) which requires a number of Koalas to carry objects (balls) from one robot arm to another. The exact locations of the arms are not known (we could suppose they are mobile), necessitating a search for each arm. This scenario could represent various real-world applications. In a military context, unmanned vehicles could be used to carry supplies across hostile territory, between manned units. Likewise in a disaster situation or a factory setting, the robots might be used to carry objects through dangerous areas. Figure 7.2 shows the scenario graphically.

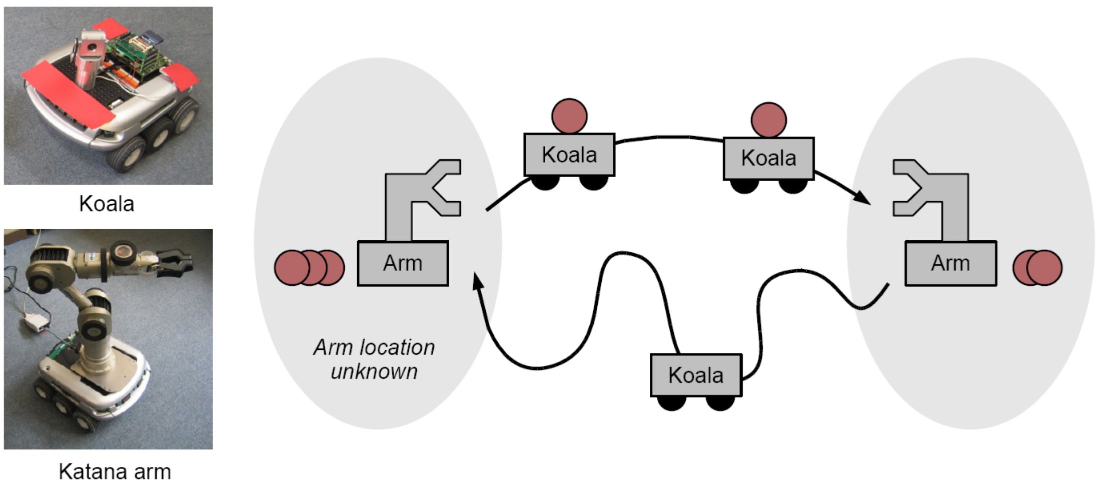


Figure 7.2: Supply scenario

The functional requirements for this application are the Surveyor (to search for the arms) and the Grabber and Placer (on the arms). The structural constraints are:

1. Obstacles should be avoided:
   $(\text{VectorMotionController}, n) \in arch \longrightarrow (\text{ObstacleAvoider}, n) \in arch$.
2. The grabbing and placing roles should be performed by different arms: $(\text{BallGrabber}, n_1) \in arch \wedge (\text{BallPlacer}, n_2) \in arch \longrightarrow n_1 \neq n_2$.
3. The BallGrabber and BallPlacer must have a KatanaArm and Webcam on the same node. In other words, there should be two (physically) different arms:
   $(\text{BallGrabber}, n_1) \in arch \longrightarrow (\text{KatanaArm}, n_1) \in arch \wedge (\text{Webcam}, n_1) \in arch$,
   $(\text{BallPlacer}, n_2) \in arch \longrightarrow (\text{KatanaArm}, n_2) \in arch \wedge (\text{Webcam}, n_2) \in arch$.

The main point of variation between the possible configurations for this scenario is in the choice of SearchPattern implementation. Although this fact is not necessarily known at design time, as component availability changes, we use it to restrict our discussion of non-functional properties to these components. Suppose each search pattern is annotated with the following NF properties:

- CircularSearchPattern: $(efficiency, 0.8)$, $(success, 0.5)$.

- ZigZagSearchPattern: $(\textit{efficiency}, 0.5)$, $(success, 0.6)$.
- RandomSearchPattern: $(\textit{efficiency}, 0.3)$, $(success, 0.9)$.

with the intended interpretation that "efficiency" is a measure of how quickly the pattern can find a target, and "success" is a measure of how often the pattern succeeds in discovering the target. For example, RandomSearchPattern has a low efficiency since it causes random motion with no regard to what areas have already been covered, while it has a high success rate since it eventually covers the whole area.

The weights for these properties are $w_{\textit{efficiency}} = 0.6$ and $w_{success} = 0.4$, reflecting the user's preference that the Koalas cross the perilous zone quickly and the positions of the arms (though they are mobile) are expected to be relatively predictable.

7.1.2 Distributed Assembly

We can now apply the distributed assembly algorithm to generate a configuration for this application. There are 4 nodes with the following component repositories:

- A Koala k, hosting {Koala, KoalaMotorController, VectorMotionController, ObstacleAvoider, WorldEdgeAvoider, StaticObstacleAvoider, ObstacleMap, CircularSearchPattern, ZigZagSearchPattern, RandomSearchPattern, BallSurveyor, VisualDoorLocator, Webcam, GoToTask, VisualLineFollower}
- Two arms a_1, a_2, each hosting {KatanaArm, BallGrabber, BallPlacer, Webcam}
- A location server s, hosting {SkyCamera}

Each node is provided with the set of functional requirements and the structural constraints. The initial state of each node is thus

$$c_0 = \{(\textbf{req}, \_, \text{Surveyor}), (\textbf{req}, \_, \text{Grabber}), (\textbf{req}, \_, \text{Placer})\}$$

The nature of gossip means that there are many possible execution sequences, however we can only consider one here. Suppose in the first step that k applies rule 2 (Section 5.2.2) to satisfy the Surveyor requirement using the BallSurveyor, and propagates the state

$$c_1 = c_0 \cup \{(\textbf{prov}, (\text{BallSurveyor}, k), \text{Surveyor}), (\textbf{req}, (\text{BallSurveyor}, k), \text{Camera}),$$
$$(\textbf{req}, (\text{BallSurveyor}, k), \text{SearchPattern})\}$$

Simultaneously, a_1 satisfies the Grabber requirement with BallGrabber, and propagates the state

$$c_2 = c_0 \cup \{(\textbf{prov}, (\text{BallGrabber}, a_1), \text{Grabber}), (\textbf{req}, (\text{BallGrabber}, a_1), \text{Camera}),$$
$$(\textbf{req}, (\text{BallGrabber}, a_1), \text{RobotArm})\}$$

In the next step, a_1 receives the state c_1 from k, and again adds the BallGrabber. k satisfies the new requirements for a SearchPattern and a Camera by adding the Webcam and choosing one of the SearchPattern implementations. The utility of each is calculated according to the user preferences:

$$U(\text{CircularSearchPattern}) = 0.8 \times 0.6 + 0.5 \times 0.4 = 0.68$$
$$U(\text{ZigZagSearchPattern}) = 0.5 \times 0.6 + 0.6 \times 0.4 = 0.54$$
$$U(\text{RandomSearchPattern}) = 0.3 \times 0.6 + 0.9 \times 0.4 = 0.54$$

The CircularSearchPattern is thus selected producing a new state

$$c_3 = c_1 \cup \{(\textbf{prov}, (\text{Webcam}, k), \text{Camera}), (\textbf{prov}, (\text{CircularSearchPattern}, k), \text{SearchPattern}),$$
$$(\textbf{req}, (\text{CircularSearchPattern}, k), \text{MotionController}),$$
$$(\textbf{req}, (\text{CircularSearchPattern}, k), \text{LocationServer})\}$$

Suppose then that a_1 receives this new state and adds the BallGrabber and the KatanaArm to produce a state

$$\begin{aligned}c_4 = c_3 \cup \{&(\textbf{prov}, (\text{BallGrabber}, a_1), \text{Grabber}), (\textbf{req}, (\text{BallGrabber}, a_1), \text{Camera}),\\ &(\textbf{req}, (\text{BallGrabber}, a_1), \text{RobotArm}), (\textbf{prov}, (\text{KatanaArm}, a_1), \text{RobotArm})\}\end{aligned}$$

When this is received by k it is accepted as a superset of the state known to k.

In the following step, s receives the state and satisfies the LocationServer requirement introduced by the CircularSearchPattern with the SkyCamera. k also satisfies the remaining requirements by adding the VectorMotionController and Koala.

Suppose now that the nodes k, s and a_1 agree on the state

$$\begin{aligned}c_5 = c_4 \cup \{(\textbf{prov}, (\text{SkyCamera}, s), \text{LocationServer}),\\ (\textbf{prov}, (\text{VectorMotionController}, k), \text{MotionController}),\\ (\textbf{req}, (\text{VectorMotionController}, k), \text{KoalaMotors}),\\ (\textbf{prov}, (\text{KoalaMotors}, k), \text{KoalaMotors})\}\end{aligned}$$

which has the remaining requirement of a Placer. The configuration described by the state is not yet complete, so the structural constraints cannot be applied. Simultaneously, a_1 and a_2 propose satisfying Placer with their local BallPlacer producing new states c_6 and c_7. These alternative configurations are now complete, but neither satisfy all of the structural constraints.

At this point, the nodes may backtrack a number of times (rule 3(a)), issuing death certificates for c_6 and c_7. However, under these conditions backtracking will be unsuccessful. Eventually exhaustive search will be used (rule 3(b)) and the ObstacleAvoider will be added, satisfying constraint 1. If both the states proposed by a_1 and a_2 are extended in this manner, then the one which will satisfy all the constraints is that in which a_2 hosts the BallPlacer. This complete and valid solution c_v will eventually propagate to all nodes, and convergence will be detected.

The chosen configuration is shown in Figure 7.3, which additionally shows which nodes host which components.

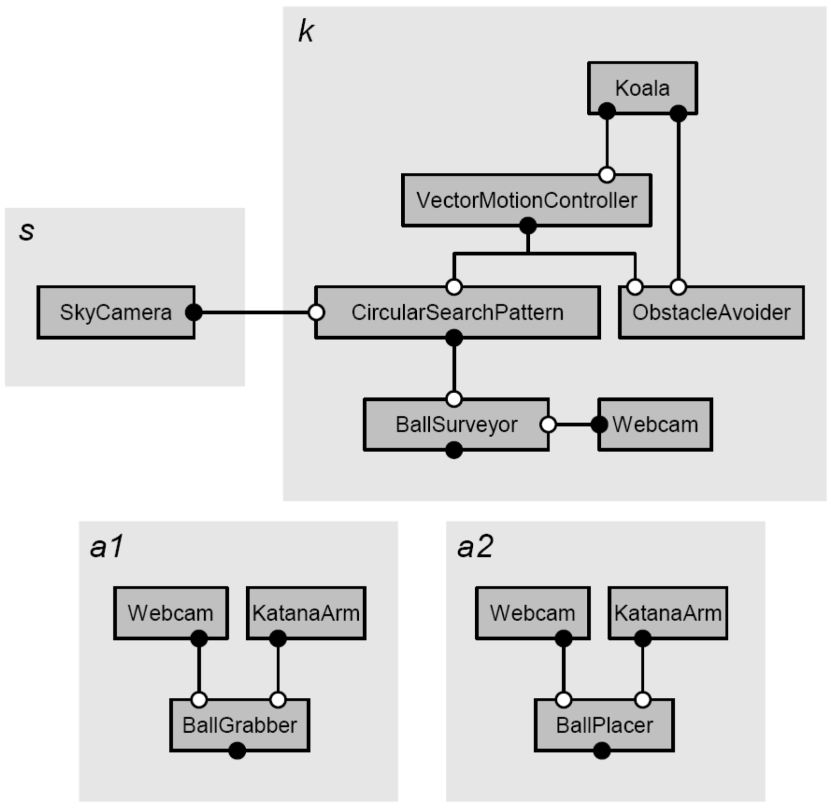


Figure 7.3: Assembled configuration for 4 nodes

Notice that the generated configuration includes a single Koala only, while the intent was to use several to accelerate the work. This can be achieved by trivially duplicating the sub-configuration for k for all k_i. It is possible to encode this duplication as a number of intricate structural constraints (similar to the way in which the Grabber and Placer roles were kept disjoint), but this is an inefficient solution both in terms of complexity presented to the designer, and in terms of the execution time, since it causes extensive backtracking and exhaustive search.

A related problem which our work does not address is that of correct and optimal allocation of which nodes are to instantiate which components (when a choice is available). The correctness of the configuration for Koalas depends on this allocation of components to nodes: (i) the ObstacleAvoider must be bound to the same instance of Koala as the VectorMotionController since the Koala must avoid the obstacles detected by *its own* sensors, (ii) the location provided by the SkyCamera and used by the CircularSearchPattern must be for the same physical robot on which the CircularSearchPattern[3] is hosted, and (iii) the Koala used (transitively) by the BallSurveyor must be hosted by the same physical node as the Webcam, since the BallSurveyor relates the image produced by the camera on a particular robot to the motion of that same robot. Where correctness is not an issue, there is also a question of optimality. Given a set of nodes hosting identical sets of components, a solution with many inter-node bindings would be inefficient in terms of communication cost, while greater efficiency could be achieved by having a preference for bindings on the same host. On the other hand, if the application requires few bindings and the user has a preference for increased performance through parallel computation, the use of more nodes might be preferred.

7.1.3 Adaptation

The above configuration can be adapted by changing the search pattern used, for example if the CircularSearchPattern fails due to a bug, or if its claimed success rate proves to be woefully inaccurate. Should this happen, the CircularSearchPattern is marked unavailable and a death certificate is issued by node k for the above configuration, and then node k backtracks. Depending on the amount of backtracking required, a new solution—in this case involving the ZigZagSearchPattern—is found more quickly than the original assembly. An interesting observation made during our experiments is that switching between search patterns is almost imperceptible to a user since the re-assembly process happens concurrently with normal execution of the non-failed components. In

[3]Although this is strictly an application-layer issue, not an architectural one.

particular, the ObstacleAvoider continues ensuring the Koala does not hit an obstacle.

7.1.4 Summary

This case study has demonstrated the use of the approach to assemble and adapt a configuration of components hosted by a number of distributed nodes which co-operate to achieve a functional goal. The scenario also had associated NF preferences and structural constraints, entailing some backtracking and exhaustive search in addition to local (incremental) NF selection. However, the few variation points of the scenario have meant that it has not exercised aggregate NF selection.

7.2 SatMotion

7.2.1 Scenario

The next scenario is based on [BHRE07] and shows how an existing problem can be translated into one of architectural reconfiguration and thence solved using our approach. SatMotion is an application designed to align satellite antennas using alternative means that are enabled by selecting different components. Figure 7.4 shows the full set of components arranged into a dependency graph. This information is expressed in [BHRE07] as architectural constraints, but its trivial translation into dependencies means that the performance cost of trying to satisfy explicit structural constraints can be avoided.

The functional requirement used to construct configurations is represented by the SatMotion interface. This is implemented at the top level by three alternatives. The alternatives for the requirements of the top-level components mean that there

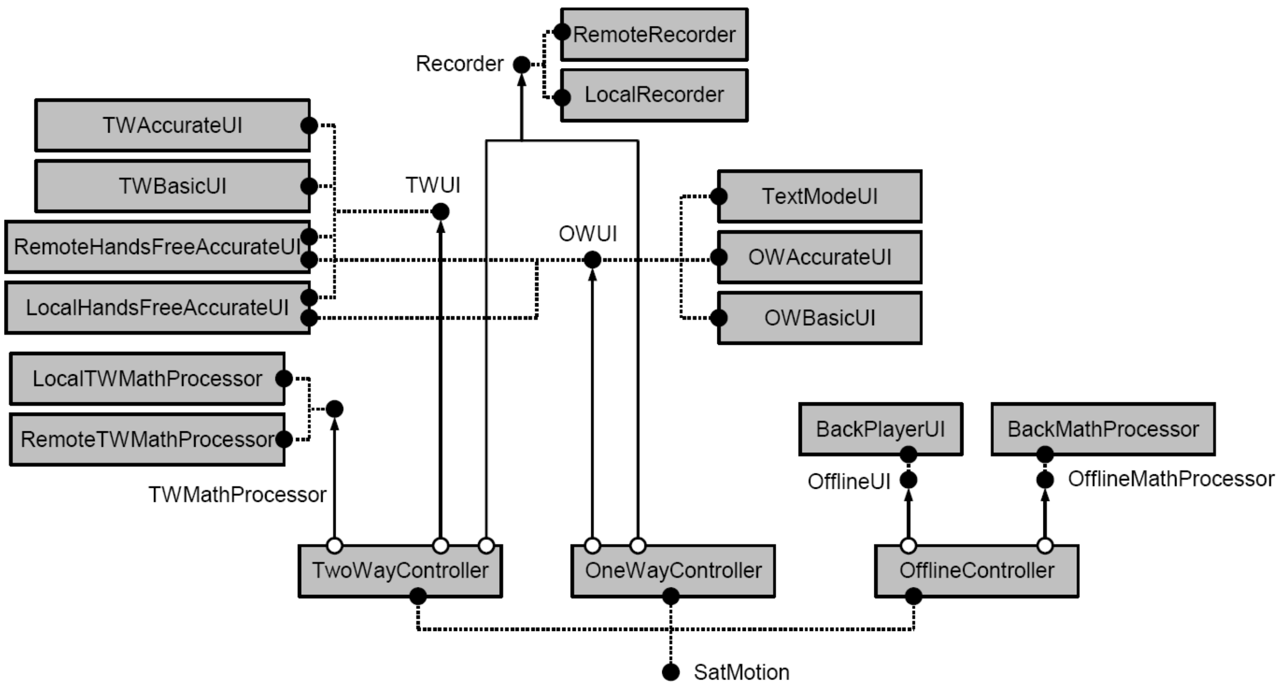


Figure 7.4: SatMotion components

are 27 solutions in total. This space of alternatives is substantially smaller than that given in [BHRE07] since we do not consider alternative locations for the remote components (of which there are 3). Indeed we regard allocation of components to nodes as a separate problem (as mentioned above). The search space may be further limited by changing component availability. For example, the OfflineController is the only viable choice when no internet connection is present. The other (less significant) difference with the prior formulation is that we cannot directly express an optional requirement (as in the case of the Recorder) except by introducing a “null” recorder.

Since we have expressed the structural constraints in dependencies, there is no need for further explicit constraints. We do however use some NF properties to choose the optimal configuration for the user’s preferences (for simplicity the utility values of each NF property are shown instead of the original values):

- TwoWayController: $(accuracy, 0.95)$.
- OneWayController: $(accuracy, 0.7)$.

- OfflineController: $(accuracy, 0.4)$.
- LocalRecorder: $(bandwidthCost, 1)$, $(performance, 0.7)$.
- RemoteRecorder: $(bandwidthCost, 0.7)$, $(performance, 0.95)$.
- LocalTWMathProcessor: $(bandwidthCost, 1)$, $(performance, 0.5)$.
- RemoteTWMathProcessor: $(bandwidthCost, 0.8)$, $(performance, 0.95)$.
- TWBasicUI: $(accuracy, 0.5)$, $(performance, 0.95)$, $(usability, 0.7)$.
- TWAccurateUI: $(accuracy, 0.9)$, $(performance, 0.5)$, $(usability, 0.6)$.
- LocalHandsFreeAccurateUI: $(bandwidthCost, 1)$, $(accuracy, 0.8)$, $(performance, 0.4)$, $(usability, 0.8)$.
- RemoteHandsFreeAccurateUI: $(bandwidthCost, 0.6)$, $(accuracy, 0.8)$, $(performance, 0.95)$, $(usability, 0.8)$.
- OWBasicUI: $(accuracy, 0.5)$, $(performance, 0.95)$, $(usability, 0.7)$.
- OWAccurateUI: $(accuracy, 0.9)$, $(performance, 0.5)$, $(usability, 0.6)$.
- TextModeUI: $(accuracy, 0.9)$, $(performance, 0.95)$, $(usability, 0.3)$.

where the annotations are relatively self-explanatory. For example, “performance” is a measure of the computational performance of the component.

Finally, we suppose that the user is primarily interested in accuracy, with performance as a second priority. This gives the weights:

$$\begin{array}{ll} w_{accuracy} = 0.4 & w_{performance} = 0.3 \\ w_{usability} = 0.2 & w_{bandwidthCost} = 0.1 \end{array}$$

Candidate solution	U_{agg}
TwoWayController, RemoteRecorder, RemoteHandsFreeAccurateUI, RemoteTWMathProcessor	0.93125
TwoWayController, LocalRecorder, RemoteHandsFreeAccurateUI, RemoteTWMathProcessor	0.92
OfflineController, BackPlayerUI, BackMathProcessor	0.92
TwoWayController, RemoteRecorder, TWAccurateUI, RemoteTWMathProcessor	0.9075
TwoWayController, RemoteRecorder, TWBasicUI, RemoteTWMath-Processor	0.90625
TwoWayController, RemoteRecorder, RemoteHandsFreeAccurateUI, LocalTWMathProcessor	0.9025
TwoWayController, RemoteRecorder, LocalHandsFreeAccurateUI, RemoteTWMathProcessor	0.9
TwoWayController, LocalRecorder, TWAccurateUI, RemoteTWMath-Processor	0.89625
TwoWayController, LocalRecorder, TWBasicUI, RemoteTWMathPro-cessor	0.895
TwoWayController, LocalRecorder, RemoteHandsFreeAccurateUI, LocalTWMathProcessor	0.89125
TwoWayController, LocalRecorder, LocalHandsFreeAccurateUI, Re-moteTWMathProcessor	0.88875
OneWayController, RemoteHandsFreeAccurateUI, RemoteRecorder	0.88667
OneWayController, TextModeUI, RemoteRecorder	0.88
TwoWayController, RemoteRecorder, TWAccurateUI, LocalTWMath-Processor	0.87875
TwoWayController, RemoteRecorder, TWBasicUI, LocalTWMathPro-cessor	0.8775
OneWayController, RemoteHandsFreeAccurateUI, LocalRecorder	0.87167
TwoWayController, RemoteRecorder, LocalHandsFreeAccurateUI, LocalTWMathProcessor	0.87125
TwoWayController, LocalRecorder, TWAccurateUI, LocalTWMathPro-cessor	0.8675
TwoWayController, LocalRecorder, TWBasicUI, LocalTWMathProces-sor	0.86625
OneWayController, TextModeUI, LocalRecorder	0.865
TwoWayController, LocalRecorder, LocalHandsFreeAccurateUI, Lo-calTWMathProcessor	0.86
OneWayController, OWAccurateUI, RemoteRecorder	0.855
OneWayController, OWBasicUI, RemoteRecorder	0.85333
OneWayController, LocalHandsFreeAccurateUI, RemoteRecorder	0.845
OneWayController, OWAccurateUI, LocalRecorder	0.84
OneWayController, OWBasicUI, LocalRecorder	0.83833
OneWayController, LocalHandsFreeAccurateUI, LocalRecorder	0.83

Table 7.1: Aggregate utility of SatMotion configurations

7.2.2 Assembly

Now we apply the centralised algorithm and use aggregate NF selection since the space of solutions is relatively small. Table 7.1 shows the full set of solutions. These are generated using the dependency analysis as described in Section 4.2.3. For example, the nineteenth in the table is found by selecting the TwoWayController to satisfy the SatMotion requirement, and then by considering the three requirements of TwoWayController. Recorder is satisfied by choosing LocalRecorder, TWUI is satisfied by choosing TWBasicUI and TWMathProcessor is satisfied by choosing LocalTWMathProcessor. This produces a complete configuration $arch$ for which the aggregate utility can be computed:

$$U_{agg}(arch) = \frac{\begin{array}{c} U(\text{TwoWayController})+ \\ U(\text{LocalRecorder})+ \\ U(\text{TWBasicUI})+ \\ U(\text{LocalTWMathProcessor}) \end{array}}{4} = 0.86625$$

The solution with the highest utility given the preferences is {TwoWayController, RemoteRecorder, RemoteHandsFreeAccurateUI, RemoteTWMathProcessor} with utility 0.93125. This makes intuitive sense, since the user weighted accuracy highly, and also performance (the remote implementations delegate the computational cost). The experiments of Section 4.4.4 suggest this solution would be derived in a small fraction of a second. Figure 7.5 displays the configuration graphically.

7.2.3 Adaptation

The situation most likely to arise is a loss of connectivity which makes a whole swathe of the components unavailable. This would leave only one solution for

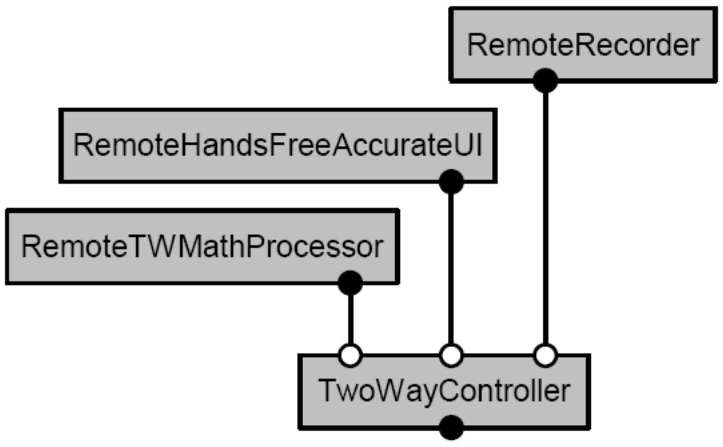


Figure 7.5: Optimal SatMotion configuration

adaptation, which is that using the OfflineController.

If however, just one connection were lost—for example, to the RemoteRecorder—then only that component would become unavailable and a new configuration would be found. The new solution would be found more rapidly than the initial assembly since approximately half of the previous solutions (those which used the RemoteRecorder) are no longer viable. Under the current preferences, the obvious solution of replacing the RemoteRecorder with the LocalRecorder has the next highest utility, although this is shared with the OfflineController solution.

7.2.4 Summary

This case study has demonstrated the use of the centralised approach for assembling and adapting a configuration of components in a scenario proposed by others. The case study has shown that it is possible to encode apparently complex structural constraints as dependencies to gain a performance benefit, thus making it feasible to use aggregate non-functional selection. In this way, the solutions chosen are optimal with respect to the preferences chosen by the user.

7.3 Qualitative Evaluation

In this section we discuss the objectives outlined in the introduction and the extent to which they have been met by the work described herein. Then we consider the significance and ramifications of certain assumptions and simplifications which have been made along the way.

7.3.1 Requirements

1. Declarative autonomy. Minimising user involvement in the specifics of the system's operation ensures that the dependency on the user's design-time assumptions is reduced. Declarative expressions of requirements, goals and preferences enhance opportunities for adaptation by not specifying detailed solutions.

The starting point of our approach is the set of functional requirements provided by the goal management layer, which derives them from an explicit functional goal. The set of functional requirements is nothing more than a list of interfaces, which means that there are potentially a huge number of solutions. This is both a benefit and a drawback since there are many opportunities for adaptation but, at the same time, searching for a solution is more expensive. The space of solutions can be constrained with further declarative information from the user: structural constraints place limits on the kinds of configurations which are acceptable without (except in degenerate cases) specifying exact configurations (as in the robotics case study), and non-functional preferences inform the assembly procedure as to which kinds of components and configurations are preferred.

2. Explicit NF properties. Explicit treatment of non-functional information can enable the system to make the best choices with respect to the properties the user wishes to see optimised.

Each of the components is annotated with a number of NF properties describing aspects such as the performance or reliability of the component. These annotations are transformed by utility functions and weights provided by the user into a utility for each component representing how closely it matches the user's preferences. This ranks components in a manner which allows the assembly procedure to choose between them. Whereas aggregate selection ranks complete configurations based on aggregate utility, incremental selection chooses between components providing the same interface while assembling a single solution. Aggregate NF selection was of particular use in the second case study in which there were many alternative configurations.

3. Explicit structural constraints. Explicit treatment of structural constraints restricts the system to producing solutions which fall into well-understood categories such as architectural styles. Structural constraints also provide a means to encode certain domain-specific rules such as mutual exclusion between a pair of components. Such constraints were of particular use in the robotics case study to handle peculiarities of the domain, such as the need for certain components to reside on the same host.

The general nature of structural constraints means that the assembly procedure can evaluate only complete solutions against them, and delegates to a general constraint checker to do so. When solutions do not meet the constraints the assembly procedure backtracks to find alternatives that do meet the constraints. Structural constraints have the effect of reducing the search space but they also remove opportunities for adaptation.

4. Safety. A modification of a running system must preserve the consistency of the state of components and thus of the application so that the modification does not cause faults.

When an adaptation has been chosen, the tranquility protocol is used to ensure the modifications are applied safely. Tranquility ensures safety by waiting for a quiescent point (at which no significant computation is being performed) to apply changes. Additionally our system has a mechanism for preserving component state information.

Tranquility was especially relevant in the robotics domain for ensuring safe thread management in the VectorMotionController and other components, and for enabling obstacle avoidance to continue execution during a reconfiguration, as mentioned above.

5. Decentralisation. Decentralisation provides the overall system with increased reliability and scalability through improved performance. Although this is true of many systems, it has particular relevance for an adaptive system since faults and wide distribution are expected to be the norm.

Our assembly procedure has been decentralised by making use of a gossip protocol. Gossip removes the reliance on a single node, and we have shown that gossip scales well to large numbers of nodes. If any node fails during assembly, the procedure continues without it, and when it restarts, it quickly receives the current solution. Decentralisation comes at some cost in absolute execution time (with respect to a centralised scheme) due to the extra requirement of reaching agreement between all the nodes, plus the unavoidable communication delays. However, agreement is reached in a logarithmic number of steps, meaning that the delay is not prohibitive, even for large systems.

Our final desire was to find techniques that provide scalability and performance appropriate for application at runtime. The scalability, in terms of nodes involved in the decentralised case, and of the size of component repositories in the centralised case, has been shown through experiments described in previous chapters. The difficulty, as regards performance, is that random backtracking can make the time to derive reconfigurations unpredictable. As described in Section 4.3.1, this can occur when the structural constraints bear no relation to the dependency graph. Finally, the two non-functional selection strategies provide a means to achieve performance appropriate to the application domain. In the robotics domain, incremental selection is the most appropriate.

In summary, our approach meets or exceeds our objectives in developing a declarative, distributable approach for assembly and adaptation of autonomous component-based systems which respects structural constraints and non-functional preferences. This approach has been shown to be effective in two rather varied case studies by addressing the domain-specific issues encountered with generalised techniques. In the next section we discuss some of our design choices, assumptions (Section 4.1) and the consequent limitations.

7.3.2 Limitations

Configurations as sets. One of the first simplifications made was to treat configurations as a set of components, and to assume the bindings (the connectors) could be derived automatically from the dependencies.

An arrangement of bindings can be found (trivially) by binding every requirement of a given interface to any provision of the same interface. When there is a single implementation of every interface, the arrangement is unique. Conversely, where the arrangement is not unique, this follows from the presence of multiple implementations of one or more interfaces. The multiple possible arrangements cannot be distinguished *functionally* since every requirement is bound to a matching provision providing equivalent functionality. However, the arrangements do differ in non-functional aspects such as performance and memory load. For example, a server component may process requests at a fixed rate. If it has two clients, they will experience degraded performance with respect to the case when two server components are used and each services one client.

Since our approach primarily uses functional requirements to assemble solutions, it does not introduce multiple implementations of a given interface (including multiple instances of a given component type) without being forced to do so by way of a structural constraint as in the robotics case study. This simplification, then, means that our technique cannot make informed choices between functionally equivalent, but non-functionally different arrangements of bindings. On the other hand, the simplification does reduce the space of solutions, providing a performance benefit.

Allocation of components to nodes. A related problem which our approach does not address is that of deciding which nodes are to host which components. Again, functionally, the different allocations are equivalent. However, they differ for reasons of correctness and non-functional characteristics.

Certain allocations may be incorrect if, for example, one component is dependent upon being located on the same physical host as another (or equally upon being located on a different physical host). This is often a relevant concern in domains which depend heavily upon hardware provision. For example, in the robotics case study, it was necessary that the BallSurveyor and Webcam reside on the same host. It is possible to enforce correctness through intricate structural constraints but this can be inefficient. An alternative technique is to partition the components into sets which must (or must not) reside on the same host. Assembly can then be performed for each host independently, and the result duplicated across all identical hosts. This divide and conquer technique should provide a substantial performance increase when compared to using structural constraints.

Apart from correctness, there are also non-functional concerns which make some allocations preferable to others. For example, having one node host all the components except one hosted by another node is likely to be vastly inefficient compared to a fairer balance of load between the nodes.

Despite these limitations, the second case study showed that the search space can be vastly reduced and hence performance can be improved by not considering which nodes should host which components.

These limitations might be overcome by sacrificing some performance in the decentralised case, where allocation is firstly influenced by which nodes are able to host which components (an input of the algorithm), and secondly by the random pattern of message exchange caused by gossip. This might be achieved by firstly dividing the problem into multiple assembly procedures for correctness, and then considering the non-functional properties of the moveable components and the various nodes to attain an optimal allocation.

The correctness of non-functional annotations. The degree to which our approach makes optimal selections is heavily dependent upon the accuracy of the non-functional information annotated on each component. However, this accuracy may be limited by the user providing incorrect information (either accidentally or maliciously) and by the fact that non-functional properties are often dependent upon the context in which they are placed. In other words, non-functional properties are not compositional and not encapsulated in the way functional properties are.

The properties for which this problem is most accentuated are those related to a finite resource such as memory and CPU attention. For example, consider a component which claims to have a high processing rate, which also requires many other components (which for simplicity are unannotated). If the user cares about performance and gives it a high weight, then this component will be chosen over an alternative which may claim a lower processing rate but has fewer requirements. In execution, the “optimal” choice may have its performance reduced as CPU attention is diverted to the many other components. In fact, its performance may be worse than the alternative (which claimed lower performance) by virtue of its having fewer dependencies.

Both the problems of an untrustworthy user and the non-compositionality of annotations can be mitigated by the use of monitoring to update annotations with more correct values which can also trigger an adaptation when the real NF property diverges from the annotation. This amounts to providing semantics for each NF property. Although this is not a perfect solution, it is in line with the aims of an autonomous system which favours information gathered at runtime to design-time analysis, which can always be undermined by incorrect assumptions.

Non-functional requirements. The non-functional information used by our approach is subsidiary to the functional requirements and the structural constraints in that it indicates preferences rather than strict requirements. This means that the NF properties of solutions cannot be constrained, which may be necessary in some domains. However, the use of preferences means that there are more opportunities for adaptation as all the functionally valid solutions can be selected.

Non-functional constraints could be provided for in a similar manner to structural constraints by applying thresholds to the NF values. Indeed, the expressiveness of the language of structural constraints could be extended to consider the NF annotations. For example, a constraint could be written so that no configuration with less than a certain reliability is accepted (a global constraint), or that no configuration with a component with an associated monetary cost over a some value is accepted.

Chapter 8

Conclusions

ALL the work described herein has been in pursuit of our overall objective to equip an autonomous, self-managing system with the ability to adapt its software architecture to overcome the unforeseen challenges posed by the operating environment. A truly autonomous system has some decision-making capacity beyond following scripts prescribed by the designer. At the same time, the designer must provide enough information in the form of goals, requirements, constraints and preferences to direct and limit the search for adaptations to those which are meaningful in the application context. This kind of autonomous system is best suited to applications which demand a "best effort" from the system, rather than those with strict real-time or safety-critical requirements.

Our three primary objectives for a declarative approach follow from this argument. The first is that greater opportunities for adaptation can be afforded by taking a declarative approach. The second is that assembly should be guided by non-functional information about components and the designer's preferences regarding NF properties. The third is that configurations should satisfy structural constraints.

The two remaining objectives follow from practical observations about the nature of the systems our approach must adapt. Firstly, that manipulating component-based systems must respect various safety constraints, and secondly that such systems are often distributed meaning a centralised adaptive framework is either inefficient or infeasible.

The final observation is that the performance of the approach must be appropriate for dynamic reconfiguration in the application domain. This means that, for example, in the robotics domain adaptations should occur within a matter of seconds.

8.1 Contributions

The key contributions of the approach we have developed are in meeting these objectives, which have not been sufficiently addressed by other proposals. This is particularly true of the fully decentralised adaptation using gossip and the strict adherence to the principle of divorcing the designer from the details of the running system. Approaches such as [Che08, GS02] specify architectural adaptations at such a level of detail that the system has few if any choices at runtime, and is hopelessly stranded should the environment evolve in a way unanticipated by the designer. This eventuality then entails a prohibitive amount of effort (and down-time) to rewrite strategies to ensure the system copes with the new circumstances. Indeed, in the extreme case, such an approach would seem to achieve little more than the separation of the application and the adaptive concerns. Additionally there is a dearth of explicitly distributed techniques. The notable work in this area is that of Georgiadis *et al.* [GMK02] which granted each node its own adaptation manager, but suffered from the same procedural approach and poor scalability owing to its use of reliable broadcast.

To summarise, our approach makes the following contributions:

Assembly process. We have developed a search algorithm that we call the assembly process (comprised of the dependency analysis, constraint checking, and non-functional selection) which assembles configurations of components based on a declaration of the functional requirements of the domain, without resorting to procedural specifications. It then narrows its space of choices by checking candidate solutions against architectural constraints written in an expressive declarative language and ranks solutions by looking at the user's preferences, expressed abstractly as numerical weights, over non-functional properties such as reliability. The approach can make use of aggregate selection to ensure optimality with respect to aggregate utility or use incremental selection which vastly improves performance for a small cost in optimality. The linear complexity of incremental selection means that solutions can be generated (from a huge space of possibilities) in a small fraction of a second.

In total, the assembly process chooses the software architecture that meets the functional and structural requirements and that has the best non-functional characteristics.

Safe adaptation. We have extended the tranquility protocol to handle multiple replacements and distribution. This ensures that adaptations are applied safely with a minimum of disruption to running components. In addition, we have provided a mechanism for preserving component state information.

Decentralisation. We have developed a decentralised technique for self-assembly which uses aggregate gossip with death certificates for failed configurations, so that a network of peer nodes can collectively decide upon a configuration, without any single node knowing the entire set of possibilities. Without backtracking, agreement is reached within logarithmic time, ensuring that the technique scales well to very large systems. Additionally it is robust to lost messages and failing nodes.

Self-assembly framework. Finally, we have provided an overall framework for declarative self-assembly in which the above constituent techniques can be applied.

If necessary, the constituents can be replaced, such as with alternative NF selection strategies or different constraint checkers.

8.2 Future Work

Of course, there remain several areas in which the work can be improved upon in future efforts, particularly with respect to the limitations discussed above and the assumptions we have made along the way. The biggest challenge relates to our treatment of configurations as sets of components, which prevents the assembly process from differentiating functionally-equivalent configurations with different arrangements of bindings. Such configurations are likely to have different (global) non-functional properties. Likewise different allocations of components to nodes result in different NF properties. A solution for this problem would have to be coupled with an increase in expressiveness and accuracy of the NF annotations, perhaps through the use of *parametric contracts* [RFB04] which take account of the context in which the component is to be placed. In addition, resource allocation [RLLS97] and explicit NF constraints [MVS07, FB98] can ensure solutions meet the user's non-functional requirements by providing NF properties with semantics [SE04].

In the broader context, there remain questions to be answered about the manner in which adaptation techniques—in all parts of the three-layer model—respond to environmental fluctuations and whether this maximises reliability or performance over the life of the system. For example, what risk is there that the environment will change repetitively, forcing adaptations to oscillate back and forth? Machine learning techniques [KPJ+06] may be relevant to extract repeating patterns from the environment and adjust the adaptive responses appropriately to maximise utility over time and avoid excessive adaptation [PGS+07]. This can also give a performance benefit as the results of the adaptation process can be recorded for reuse when the same situation arises again.

In the goal management layer, there are opportunities for improving the planning techniques used. For example, the reactive planning algorithm used in our reification does not account for non-functional aspects which may be relevant at that level. Recall that when an environment state has two exiting transitions, the one lying on the path that is shorter in the number of transitions is chosen. A very immediate benefit can be achieved by making that choice on a different basis such as a probability which indicates the reliability of the transition. In this manner, the reactive plan would include the most probable paths to the goal.

The performance of planning is also highly dependent upon the size of the state space, or in other words, upon the level of abstraction used. At the lowest level, the planning domain is determined by the semantics of the interfaces available in the component repository. Explicit treatment of semantics would enable specification-based component selection (Section 2.3.11) to be used. However, it is likely to be infeasible to perform planning directly at this level of abstraction. At the highest level of abstraction is the user's goal, and so it becomes necessary to elaborate the goal in the direction of the component semantics. This is partially addressed by the hierarchy of planning domains given in [SHMK07], but is part of a larger question about how the user's requirements including the functional goal, architectural constraints and the non-functional requirements and preferences are elicited in a precise and expressive way.

It is a substantial challenge to develop techniques which respect the user's many and diverse concerns for the best architecture given the foreseeable circumstances, which remains the best architecture come what may. If the challenge can be overcome, then the autonomous systems being progressively deployed in business, military, humanitarian scenarios and even the home, can benefit from independence of choice and strengthened reliability, and leave the user to his freedom.

Bibliography

[ADG98] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 21–37, 1998.

[AG94] R. Allen and D. Garlan. Formalizing architectural connection. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, pages 71–80, 1994.

[AHE$^+$06] M. Alia, G. Horn, F. Eliassen, M.U. Khan, R. Fricke, and R. Reichle. A component-based planning framework for adaptive systems. *LECTURE NOTES IN COMPUTER SCIENCE*, 4276:1686, 2006.

[AHW04] Naveed Arshad, Dennis Heimbigner, and Alexander L. Wolf. A planning based approach to failure recovery in distributed systems. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 8–12, New York, NY, USA, 2004. ACM Press.

[AHW07] Naveed Arshad, Dennis Heimbigner, and Alexander L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Journal*, 15(3):265–281, 2007.

[Ale79] C. Alexander. *The Timeless Way of Building*. Oxford University Press, USA, 1979.

[Bat00] John Bather. *Decision Theory: An Introduction to Dynamic Programming and Sequential Decisions*. Wiley, 2000.

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. In *Software Product Line Conference*, 2005.

[Bat06] Don Batory. Feature interactions in feature-based program synthesis. *University Of Texas*, 2006.

[BB09] Nelly Bencomo and Gordon Blair. Using architecture models to support the generation and operation of component-based adaptive systems. In *SEFSAS*. Springer, 2009.

[BCDLP03] P. Bertoli, A. Cimatti, U. Dal Lago, and M. Pistore. Extending PDDL to nondeterminism, limited sensing and iterative conditional plans. In *Proceedings of ICAPS'03 Workshop on PDDL*. Citeseer, 2003.

[BGF$^+$08] N. Bencomo, P. Grace, C. Flores, D. Hughes, and G. Blair. Genie: Supporting the model driven development of reflective, component-based adaptive systems. In *Proceedings of the 30th international conference on Software engineering*, pages 811–814. ACM New York, NY, USA, 2008.

[BHRE07] G. Brataas, S. Hallsteinsen, R. Rouvoy, and F. Eliassen. Scalability of decision models for dynamic product lines. *Int. Workshop on Dynamic Software Product Line (DSPL)*, pages 1–10, 2007.

[BLHM02] Don Batory, Roberto E. Lopez-Herrejon, and Jean-Philippe Martin. Generating product-lines of product-families. In *ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering*, page 81, Washington, DC, USA, 2002. IEEE Computer Society.

[BLM08] Roberto Bruni, Alberto Lluch Lafuente, and Ugo Montanari. Hierarchical design rewriting with maude. In *7th International Workshop on Rewriting Logic and its Applications (WRLA'08)*, 2008.

[BLMR04] A.K. Bandara, E.C. Lupu, J. Moffett, and A. Russo. A goal-based approach to policy refinement. In *Proceedings of the 5th IEEE Workshop on Policies for Distributed Systems and Networks*. Citeseer, 2004.

[CEM03] R. Chatley, S. Eisenbach, and J. Magee. Modelling a framework for plugins. In *Specification and verification of component-based systems*. Citeseer, 2003.

[CGI$^+$08] Betty H. C. Cheng, Holger Giese, Paola Inverardi, Jeff Magee, and Rogério de Lemos. Software engineering for self-adaptive systems: A research road map. In *Software Engineering for Self-Adaptive Systems*, 2008.

[CGS$^+$02] Shang-Wen Cheng, David Garlan, Bradley R. Schmerl, Joao Pedro Sousa, Bridget Spitnagel, and Peter Steenkiste. Using architectural style as a basis for system self-repair. In *WICSA 3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 45–59, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[Che08] Shang-Wen Cheng. *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*. PhD thesis, Carnegie Mellon University, 2008.

[Dev95] J. L. Devore. *Probability and Statistics or Engineering and the Sciences*. Duxbury Press, 1995.

[DGH$^+$87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of Distributed Computing*, pages 1–12. ACM New York, NY, USA, 1987.

[Dug05] Dominic Duggan. Type-based hot swapping of running modules. *Acta Informatica*, 41(4):181–220, 2005.

[DvdHT02] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 21–26, New York, NY, USA, 2002. ACM Press.

[EGMT09] Ilenia Epifani, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. Model evolution by runtime adaptation. In *ICSE 2009*, 2009.

[FB98] X. Franch and P. Botella. Putting non-functional requirements into software architecture. *Software Specification and Design, 1998. Proceedings. Ninth International Workshop on*, pages 60–67, 1998.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 175–188, New York, NY, USA, 1994. ACM Press.

[Gat98] E. Gat. Three-Layer Architectures. *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, pages 195–210, 1998.

[GBJC07] Antônio Tadeu A. Gomes, Thais V. Batista, Ackbar Joolia, and Geoff Coulson. *Architecting Dynamic Reconfiguration in Dependable Systems*. Springer, 2007.

[GM03] Dimitra Giannakopoulou and Jeff Magee. Fluent Model Checking for Event-Based Systems. *European Software Engineering Conference / Foundations of Software Engineering (ESEC/FSE)*, 2003.

[GMK02] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising software architectures for distributed systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 33–38, New York, NY, USA, 2002. ACM Press.

[GMW00] D. Garlan, R.T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. *Foundations of Component-Based Systems*, 2000.

[GPSS04] David Garlan, Vahe Poladian, Bradley Schmerl, and Joao Pedro Sousa. Task-based self-adaptation. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 54–57, New York, NY, USA, 2004. ACM Press.

[GS02] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA, 2002. ACM Press.

[GT99] F. Giunchiglia and P. Traverso. Planning as Model Checking. *European Conference on Planning*, 1999.

[GT04] John C. Georgas and Richard N. Taylor. Towards a knowledge-based approach to architectural adaptation management. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 59–63, New York, NY, USA, 2004. ACM Press.

[Hei05] G.T. Heineman. *Component-Based Software Engineering*. Springer, 2005.

[HSMK09] William Heaven, Daniel Sykes, Jeff Magee, and Jeff Kramer. A Case Study in Goal-Driven Architectural Adaptation. *Software Engineering for Self-Adaptive Systems*, 2009.

[IMTA05] P. Inverardi, L. Mostarda, M. Tivoli, and M. Autili. Synthesis of correct and distributed adaptors for component-based systems: an automatic approach. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, page 409. ACM, 2005.

[Jac95] M. Jackson. The world and the machine. In *Proceedings of the 17th international conference on Software engineering*, pages 283–292. ACM, 1995.

[JMB05] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems (TOCS)*, 23(3):252, 2005.

[KC03] JO Kephart and DM Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[KDG03] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE*, volume 44, pages 482–491. IEEE COMPUTER SOCIETY PRESS, 2003.

[KM90] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.

[KM98] J. Kramer and J. Magee. Analysing dynamic change in distributed software architectures. In *IEE Proceedings*, volume 145, pages 146–154, 1998.

[KM07] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. *FOSE*, 0:259–268, 2007.

[KPJ$^+$06] D. Kim, S. Park, Y. Jin, H. Chang, Y.S. Park, I.Y. Ko, K. Lee, J. Lee, Y.C. Park, and S. Lee. SHAGE: A framework for self-managed robot software. *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, pages 79–85, 2006.

[Lis87] Barbara Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on Object-oriented* *programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, 1987. ACM.

[LKA$^+$95] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, W. Mann, and D. Bryan. Specification and analysis of system architecture using rapide. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, pages 336–355, 1995.

[LM98] D. Le Metayer. Describing software architecture styles using graph grammars. *Software Engineering, IEEE Transactions on*, 24(7):521–533, 1998.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[MAD06] MADAM. MADAM: Theory of adaptation (technical report). `http://www.intermedia.uio.no/display/madam/D2.2+-+Theory+of+Adaptation`, 2006.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.

[MDSR07] A. Mukhija, A. Dingwall-Smith, and D.S. Rosenblum. Qos-aware service composition in dino. In *Proceedings of the Fifth European Conference on Web Services (ECOWS'07)-Volume 00*, pages 3–12. IEEE Computer Society Washington, DC, USA, 2007.

[MG99] Kaveh Moazami-Goudarzi. *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*. PhD thesis, Imperial College, 1999.

[MGK96] Kaveh Moazami-Goudarzi and Jeff Kramer. Maintaining node consistency in the face of dynamic change. *ICCDS*, 00:62, 1996.

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, New York, NY, USA, 1996. ACM Press.

[MK00] Jeff Magee and Jeff Kramer. *Concurrency: State Models & Java Programming*. Wiley, 2000.

[MKM06] Andrew McVeigh, Jeff Kramer, and Jeff Magee. Using resemblance to support component reuse and evolution. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, pages 49–56, New York, NY, USA, 2006. ACM Press.

[MLGI05] S.B. Mokhtar, J. Liu, N. Georgantas, and V. Issarny. QoS-aware dynamic service composition in ambient intelligence environments. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 317–320. ACM New York, NY, USA, 2005.

[MMM97] Rym Mili, Ali Mili, and Roland T. Mittermeir. Storing and retrieving software components: A refinement based system. *IEEE Trans. Softw. Eng.*, 23(7):445–460, 1997.

[MTWJ96] N. Medvidovic, R.N. Taylor, and E.J. Whitehead Jr. Formal modeling of software architectures at multiple levels of abstraction. *California Software Symposium*, 714:824–2776, 1996.

[MUS07] MUSIC. MUSIC: Initial research results on mechanisms and planning algorithms for self-adaptation (technical report). `http://www.ist-music.eu/MUSIC/docs/MUSIC-D12-AdaptationInitialResults.pdf/view`, 2007.

[MVS07] P. Manolios, D. Vroon, and G. Subramanian. Automating component-based system assembly. In *Proceedings of the 2007 international symposium on Software testing and analysis*, page 72. ACM, 2007.

[NR07] Dirk Niebuhr and Andreas Rausch. A concept for dynamic wiring of components. In *SAVCBS 2007*, September 2007.

[OGT+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[OMT98] P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 1998 (20th) International Conference on Software Engineering*, pages 177–186, 1998.

[PA99] John Penix and Perry Alexander. Efficient specification-based component retrieval. *Automated Software Engineering*, 6:139–170, 1999.

[PGS+07] V. Poladian, D. Garlan, M. Shaw, B. Schmerl, J.P. Sousa, and M. Satyanarayanan. Leveraging resource prediction for anticipatory dynamic configuration. In *Proceedings of the First IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO-2007*, 2007.

[Pit87] B. Pittel. On spreading a rumor. *SIAM Journal on Applied Mathematics*, pages 213–223, 1987.

[PW92] D.E. Perry and A.L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT SOFTWARE ENGINEERING NOTES*, 17(4):40, 1992.

[RE94] M. Radestock and S. Eisenbach. What do you get from a π-calculus semantics? In *PARLE'94 Parallel Architectures and Languages Europe*, pages 635–647. Springer, 1994.

[RFB04] R.H. Reussner, V. Firus, and S. Becker. Parametric performance contracts for software components and their compositionality. In *Proceedings of the 9th International Workshop on Component-Oriented Programming (WCOP 04)*, pages 40–49. Citeseer, 2004.

[RHR96] Jason E. Robbins, David M. Hilbert, and David F. Redmiles. Using critics to analyze evolving architectures. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, pages 90–93, New York, NY, USA, 1996. ACM Press.

[RLLS97] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for QoS management. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 298–307, 1997.

[SB98] Yannis Smaragdakis and Don S. Batory. Implementing layered designs with mixin layers. In *ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 550–570, London, UK, 1998. Springer-Verlag.

[Sch87] M.J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *IJCAI*, volume 87, pages 1039–1046. Citeseer, 1987.

[SDK$^+$95] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, pages 314–335, 1995.

[SE04] R. Staehli and F. Eliassen. Compositional quality of service semantics. In *SAVCBS 2004 Specification and Verification of Component-Based Systems*, page 62, 2004.

[SFLD^{+}07] A. Schaeffer-Filho, E. Lupu, N. Dulay, S. L. Keoh, K. Twidle, M. Sloman, S. Heeps, S. Strowes, and J. Sventek. Towards supporting interactions between self-managed cells. In *Proceedings of the 1st IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 224–233, Boston, USA, July 2007.

[SG96] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1996.

[SG02] Joao Pedro Sousa and David Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *WICSA 3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 29–43, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[SHMK07] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. Plan-Directed Architectural Change For Autonomous Systems. *Proc. of ESEC/FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, 2007.

[SHMK08] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. From Goals to Components: A Combined Approach to Self-Management. *Proc. of ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2008.

[SHMK10] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. Exploiting non-functional preferences in architectural adaptation for self-managed systems. In *Proceedings of the 2010 ACM symposium on Applied Computing*. ACM, 2010.

[Szy98] C. Szyperski. *Component Software: Beyond Object-oriented Programming*. Addison-Wesley Professional, 1998.

[TMA+95] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead Jr, and J.E. Robbins. A component- and message-based architectural style for gui software. In *Proceedings of the 17th International Conference on Software Engineering*, pages 295–304. ACM New York, NY, USA, 1995.

[vdS07] T. van der Storm. Generic feature-based software composition. *Proceedings of 6th International Symposium on Software Composition*, 2007.

[VEBD06a] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. An alternative to quiescence: Tranquility. *Proc. 22nd IEEE Int'l Conf. Software Maintenance (ICSM'06), Oct*, 2006.

[VEBD06b] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. An alternative to quiescence: Tranquility. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 73–82, Washington, DC, USA, 2006. IEEE Computer Society.

[VEHK05] Christopher Van Eenoo, O. Hylooz, and K.M. Khan. Addressing non-functional properties in software architecture using adl. *The Sixth Australasian Workshop on Software and System Architectures (AWSA 2005)*, 2005.

[vR] Robbert van Renesse. Epidemic protocols (or, gossip is good). `http://www.cis.cornell.edu/iai/events/Gossip_Tutorial.pdf`.

[Wer99] Miguel Wermelinger. *Specification of Software Architecture Reconfiguration*. PhD thesis, Universidade Nova de Lisboa, 1999.

[WSKW06] Ian Warren, Jing Sun, Sanjev Krishnamohan, and Thiranjith Weerasinghe. An automated formal approach to managing dynamic reconfiguration. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 37–46, Washington, DC, USA, 2006. IEEE Computer Society.

[WTKD04] WE Walsh, G. Tesauro, JO Kephart, and R. Das. Utility functions in autonomic systems. *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 70–77, 2004.

[YS97] D.M. Yellin and R.E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):292–333, 1997.

[ZC05] Ji Zhang and Betty H. C. Cheng. Specifying adaptation semantics. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[ZC06a] Ji Zhang and Betty Cheng. Modular model checking of dynamically adaptive programs. Technical report, Michigan State University, 2006.

[ZC06b] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 371–380, New York, NY, USA, 2006. ACM Press.

[ZC07] Ji Zhang and Betty Cheng. Re-engineering legacy systems for assured dynamic adaptation. In *Workshop on Modeling in Software Engineering (MiSE'07)*, 2007.

[ZW97] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Trans. Softw. Eng. Methodol.*, 6(4):333–369, 1997.

[ZYCM04] Ji Zhang, Zhenxiao Yang, Betty Cheng, and Philip McKinley. Adding safeness to dynamic adaptation techniques. In *ICSE04*, 2004.

Appendix A

Syntax

A.1 Syntax Of Non-Functional Annotations

Component ::= **component** *ComponentName* **[** *NFProperty⋆* **]** { *Port+* }

NFProperty ::= *PropertyName* **=** *PropertyValue* **,**

Port ::= (**prov** | **req**) *PortName* **:** *InterfaceName* **;**

A.2 Syntax Of FLTL

The grammar given here for fluent linear temporal logic in LTSA is defined in [GM03].

Fluent ::= **fluent** *FluentName* **= <** { *ActionName⋆* } **,** { *ActionName⋆* } **>**
(**initially** (**true** | **false**))?

Constraint ::= **constraint** *ConstraintName* **=** *LTLFormula*

LTLFormula ::= **true** | **false** | *FluentName* | *ActionName* |
(*LTLFormula* **)** | *UnOp LTLFormula* | *LTLFormula BinOp LTLFormula*

UnOp ::= **[]** | **<>** | **!** | **X**

BinOp ::= **U** | **W** | **&&** | **||** | **->** | **<->**