An Open Framework for Dynamic Reconfiguration

Jamie Hillman & Ian Warren
Computing Department
Lancaster University
Lancaster UK
j.hillman | iw @comp.lancs.ac.uk

Abstract

Dynamic reconfiguration techniques appear promising for building systems that have requirements for adaptability and/or high availability. Current systems that support dynamic reconfiguration tend to use a single, fixed, reconfiguration algorithm to manage the change process. Furthermore, existing change management systems lack support for measuring the impact of reconfiguration on a running system. In this paper, we introduce OpenRec, an open framework for managing dynamic reconfiguration which addresses these drawbacks. Using OpenRec, developers can observe the costs, in terms of time and disturbance, associated with making a particular run-time change. In addition, OpenRec employs an extensible set of reconfiguration algorithms where one algorithm can be substituted for another. Developers can thus make an informed decision as to which algorithm to use based on comparative analysis. Finally, OpenRec is itself dynamically reconfigurable.

1 Introduction

Dynamic reconfiguration is the process of making changes to an executing system without requiring that the system be temporarily shut down. Two classes of system for which dynamic reconfiguration is an enabling technology include adaptive systems and those with a requirement for high availability. The former operate in unpredictable environments and must adapt their behaviour according to environmental changes. Highly available systems are those which must run for extended periods and which must accomodate change during those periods.

Mobile and ubiquitous systems are an emerging example of adaptive system which operate in conditions of fluctuating resource availability [17]. For example, a mobile device might discover a new resource with which it would like to communicate. The mobile device might need to first reconfigure itself by switching to a communication protocol which is compatible with that of the discovered resource. More generally, IBM's initiative on autonomic computing [9] demands adaptive behaviour for self-configuring, self-healing, self-optimising, and self-protecting systems.

Systems with a requirement for high availability, similarly to adaptive systems, span multiple domains. For example, part of the International Space Station's control system is tasked with regulating the oxygen supply to crew members. This, along with a telecommunications switching system, needs to be highly available for safety reasons. In other cases, the motivation for high availability might be economic, such as with e-commerce and banking systems.

Ad hoc approaches to dynamic reconfiguration are used in practice but these tend to be system specific and therefore limited. For example, to upgrade a web server, incoming requests can be switched to a temporary server for the duration of the upgrade. While this is a workable technique, it is restricted to a subset of distributed client/server systems.

To provide more generic and system-independent capabilities, work in the areas of software architecture and dynamic reconfiguration has led to change management systems that operate on component-based systems. Researchers have developed reconfiguration algorithms, each with their own run-time characteristics and constraints governing their applicability, which aim to carry out change in a way that preserves system integrity. However, existing change management systems typically conform to the black-box philosophy by encapsulating a single and fixed reconfiguration algorithm. This means that developers are forced to use the algorithm of a particular change management system and cannot exploit alternative algorithms that may be better suited to a given application. We argue that the change management system should be open, giving developers choice over which algorithm to use.

Beyond conforming to the black-box philosophy, existing change management systems do not provide adequate support for measuring the cost of making run-time changes. Of interest to developers of reconfigurable systems is the cost incurred by using a particular algorithm to manage a particular change scenario for a particular system. Cost can be measured in terms of the time taken for reconfiguration and the degree of disruption experienced by components. In the absense of such quantifiable data, developers can only guess at the impact of performing a run-time change on a system's non-functional attributes, such as availability, response-time, and throughput. In addition, existing systems generally offer no means for comparing different algorithms.

Moreover, existing change management systems are not typically themselves reconfigurable. Where the management system is tightly integrated with the execution of the applications it manages, updating the management system invariably requires that it and all managed applications be shut down prior to updating it. Since change is a natural phenomenon of all software systems [14], the underlying change management system should itself be reconfigurable so that it can adapt and be evolved dynamically in a similar manner to its hosted applications.

In this paper, we introduce OpenRec, a framework for managing reconfiguration which addresses the current limitations of work to date. In short, the novel contributions made by OpenRec are:

- *Support for multiple extensible reconfiguration algorithms.* OpenRec allows multiple reconfiguration algorithms to be used to manage reconfiguration. In addition, the set of algorithms is extensible, allowing developers to add their own algorithms that better suit specific properties of their applications.
- *Support for measuring the cost of reconfiguration.* OpenRec is designed with plugin points that allow developers to monitor reconfiguration and to gather statistics relating to the degree of disturbance and time taken to carry out a reconfiguration.
- *Reconfiguration of OpenRec itself.* The OpenRec framework is itself designed to be reconfigurable. OpenRec, similarly to the applications it manages, is a component-based system with an open implementation. Reconfiguration algorithms can be substituted at run-time as can other key services.

To prime the reader with OpenRec's underlying technologies, we continue in Section 2 with a brief description of components, reflection, and dynamic reconfiguration. In Section 3, we present an overview of OpenRec, focusing on its reflective component model, its open architecture, and the benefits these bring to dynamic change management. In Section 4 we show how we have used OpenRec to reconfigure and monitor a simple EPOS (electronic point of sale) simulation system. We comment on other significant work in Section 5 before drawing conclusions and identifing avenues for further work in Section 6.

2 Core technologies

2.1 Components

A component is a unit of composition with well-defined provided and required interfaces [22]. Since components are compositional, they are intended to be assembled to form a configuration of components. Similarly to an object, a component may encapsulate state and its provided interfaces define services the component offers to others. Components are distinguished from objects in that they are *configuration independent*, which is defined by:

- *Explicit required interfaces.* In addition to provided interfaces, components define required interfaces which represent the services they require from other components. A component's required interfaces are bound at run-time to other components' provided interfaces.
- *Interconnection independence.* Components do not contain implicit references to other components. Instead, all interaction between components is handled by a connector which binds required interfaces to provided interfaces. Common types of connector include those with messaging, (remote) invocation and publish/subscribe semantics.
- *Conformance to a binary standard.* Components can be separately compiled and composed into a configuration without access to their source code.

Closely related to components is the notion of component frameworks, which Szyperski [22] defines as "collections of rules and interfaces that govern the interaction of a set of components plugged into them." In essence, a component framework is a reusable architecture which provides a means of enforcing architectural properties. Such architectural properties may relate to components, connectors and connectivity. For example, components may or may not be allowed to encapsulate state, connectors may be required to implement particular semantics, and connectivity invariants may be specified. Component frameworks are a similar concept to architectural styles [21] from the software architecture community.

Component technology is also well suited to realising reconfigurable systems. Configuration independence means that components are inherently loosely coupled and any dependencies are explicit and visible which eases the process of changing a configuration at run-time. In general, a component configuration can be manipulated dynamically by adding, removing and replacing components and changing connectivity.

2.2 Reflection

Reflection is the ability for a system to reason about and act upon itself [12]. Reflective systems comprise two levels, a base level and a meta level, that are causally connected. A change in the base layer thus causes an automatic change in the meta level and vice versa. The motivation for reflection, in all areas where it has been applied which include middleware [3] and component models [6], is to promote openness and flexibility.

Of particular relevance to our work is the integration of reflection and components. With reflective component models, the base-level corresponds to application components which are augmented with meta-level interfaces. The meta-level interfaces can be invoked at run-time and offer two complementary forms of reflection:

- *Structural reflection*. This is concerned with exposing the structure of components and connectors that realise a system. Meta-level interfaces provide operations which allow the interfaces a given component provides and requires to be inspected. In addition, operations are provided which allow the connectivity structure to be traversed. Beyond inspecting a system's structure, the meta-level interfaces also allow this structure to be modified.

- *Behavioural reflection*. This form of reflection is concerned with system activity, such as component interaction. Interceptors are typically used to listen on connectors for inter-component communications and to take action for purposes such as monitoring or accounting.

Reflective component models bring openness and self-awareness to component-based systems. Using structural reflection, a component's neighbours can be discovered, its interfaces can be identified and other meta information such as whether a particular component encapsulates critical state can be found. In addition, structural reflection offers the necessary primitives for making structural changes to a configuration.

2.3 Dynamic reconfiguration

Although reflective component models offer a powerful means for dynamic change, they offer only limited safeguards and do not guarantee that run-time changes will leave systems with their integrity intact. Two fundamental issues concerned with preserving system integrity during periods of dynamic reconfiguration are:

- *Synchronising change with the executing system*. Arbitrary changes to a configuration will likely compromise its integrity. For example, using its meta interface a component might be disconnected from its neighbours. If the disconnection is performed once the component has started to process a request on behalf of one of its neighbours but before it has returned a response, the neighbouring component might wait indefinetely for the response. Synchronisation is thus necessary to carry out change safely.

- *Preserving persistent state*. Components may encapsulate state that must survive reconfiguration. In another scenario involving replacing a component, for example, the replacement component might need access to the state encapsulated by the original component. This is invariably the case for any component whose state is subject to change since changes can influence its future behaviour. A simple example is a component which generates unique identifiers; where this component is to be replaced, the replacement component relies on knowledge of which identifiers have already been generated in order to continue to create unique identifiers. Naive use of meta interfaces could easily result in such state being discarded at run-time, with the consequence that system integrity is compromised.

Dynamic reconfiguration is concerned with bringing order to the change process with the aim of maintaining a system's integrity. In general, this is achieved using an algorithm which examines a set of proposed configuration changes and identifies a subset of of the configuration to direct towards a safe state where change is deemed synchronised with the executing system. The safe state may be reached when a component involved in reconfiguration is prevented from processing incoming requests and from initiating further requests on others. Having reached the safe state, an algorithm typically effects configuration changes and may take action to preserve persistent state before allowing the configuration of components to resume normal execution.

Reconfiguration algorithms differ in terms of behaviour and applicability constraints. To illustrate behavioural differences, algorithms can be classified according to two dimensions:

- *Approach to reaching the safe state*. Static algorithms, for example [13] [23], use knowledge of the configuration structure to identify components to make safe. As such, static algorithms will always identify the same set of components for a particular system reconfiguration. In contrast, dynamic algorithms, including [20] [4] [11], use run-time information such as component interaction to determine the set of components to make safe. In many cases, dynamic algorithms disrupt less of a system than static algorithms since they make safe only those components that are *actually* communicating with a component involved in change. Static algorithms, on the other hand, make safe the set of components that might *potentially* communicate with a component involved in reconfiguration.

- *Unit of disturbance*. Algorithms either manipulate components or connectors. Algorithms of the former type, including [13] [11], are relatively coarse-grained since they halt components, thereby preventing them from executing further. Connector-blocking algorithms, such as [23] [24], allow components to continue to execute, albeit with degraded functionality using their unblocked connectors. Algorithms that block connectors therefore impact less on an executing system.

Common constraints on algorithms include prescribed communication paradigms, the need for components to implement change management interfaces, and prerequisite knowledge of component properties. For example, a number of algorithms have been developed for distributed objects[1] ([1] [20] for CORBA and [4] for Java RMI). These algorithms therefore assume method invocation as the communication paradigm. Kramer and Magee's algorithm [13] is less restrictive in that it allows components to interact using transactions, where a transaction is defined as one or more message exchanges between two or more components. However, Kramer and Magee's algorithm requires that components implement a change management interface correctly in order for the safe state to be reached.

To illustrate prerequisite knowledge, Chen's algorithm [4] distinguishes between mutator and selector operations by allowing mutator requests to complete execution and aborting selector requests. The rationale in this case is to force a component to reach the safe state more quickly allowing only state modifying operations to complete. Mutator operations must be allowed to complete in the interest of component integrity. Once the component has completed executing any mutator requests, it is deemed to have synchronized. To implement this behaviour, the algorithm needs knowledge of which of a component's operations are mutators and which are selectors.

The above discussion exposes some fundamental differences in reconfiguration algorithms found in the literature. We argue that given such differences, algorithms will vary in terms of the costs they impose on an executing system. In addition, developers need to be aware of the constraints governing applicability of any algorithm. Hence, a framework which allows multiple algorithms to be modelled, implemented, and compared is highly desirable in determining the most appropriate algorithm to use, given a particular application.

[1]Despite being developed for distributed object systems, these algorithms can be used to manage reconfiguration of components where connectors implement method invocation semantics.

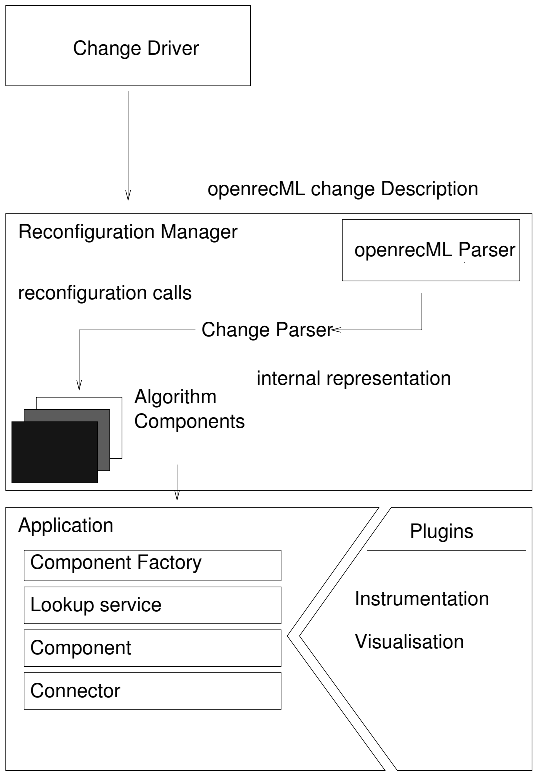


Figure 1. Framework architecture

3 Framework overview

Figure 1 introduces the architecture of our framework and shows that is split into three conceptual layers: *Change Driver*, *Reconfiguration Manager*, and *Application*. Each layer is constructed using the OpenRec reflective component model, the key classes and interfaces of which are shown in Figure 2.

For highly available systems, where change is generally initiated by a third party, change requests are submitted in the form of OpenRecML scripts to the Change Driver. OpenRecML is an XML-based configuration language. In the case of adaptive systems, the Change Driver determines if and when a change is necessary. In either case, the Change Driver forwards change requests to the Reconfiguration Manager, which is responsible for changing the running application using a reconfiguration algorithm. The Application layer comprises the configuration of components and connectors that comprise the application. In addition,

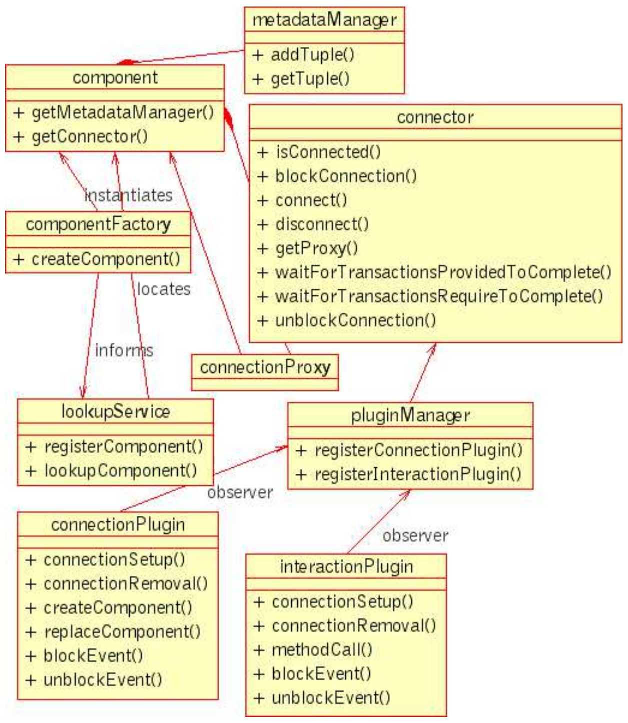
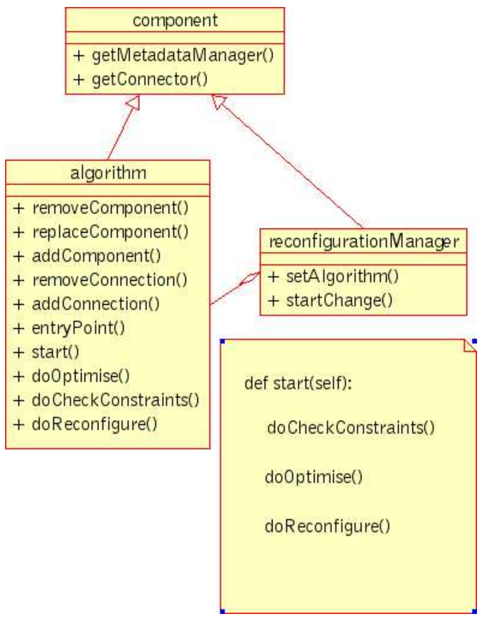


Figure 2. OpenRec component model



```
def start(self):

    doCheckConstraints()

    doOptimise()

    doReconfigure()
```

Figure 3. Algorithm components

the Application layer provides a point for plugging in monitoring components.

3.1 Support for multiple algorithms

OpenRec allows developers to configure the Reconfiguration Manager with one of several reconfiguration algorithms. In addition, developers are able to observe the behaviour of different algorithms and consequently the costs associated with using them. Reconfiguration costs are measured in terms of the number of components affected by an algorithm, the way in which components are affected, the duration of these effects and the total time required to make the change.

Each reconfiguration algorithm is implemented according to the Strategy and Template Method design patterns [8]. Figure 3 shows that the `reconfigurationManager` component plays the role of the context according to the Strategy pattern and `algorithm` the strategy. Strategy algorithms are intended to inherit from `algorithm` and override all methods with the exception of `start`. `algorithm`'s `start` method is a template method which enforces ordering of reconfiguration activities by calling the `do` methods that must be implemented by `algorithm` subclasses.

In response to a reconfiguration request from the Change Driver, the `reconfigurationManager` delegates the request to its `algorithm` instance by invoking its structural change methods. Based on this knowledge, when the `algorithm`'s `start` method is called, the algorithm is aware of which components are involved in reconfiguration and can check that any necessary constraints hold true. To do this, the `algorithm` exploits the self-awareness property of application components using their meta-level interfaces. A `metaDataManager` instance maintains a list of tuples containing meta information for a particular `component`. For example, an implementation of Kramer and Magee's algorithm [13] uses each component's `metaDataManager` to determine whether its component implements the required change management interface.

Having ensured that all constraints are satisfied, the algorithm proceeds to carry out any optimisation. In many cases, reconfiguration actions can be carried out in parallel and so this step is intended for such purposes. Following optimisation, the algorithm continues by synchronising the intended change with the application prior to making the necessary structural changes. For synchronisation, the algorithm invokes any algorithm-specific interfaces that components are known to export, as determined in the constraint checking step, in addition to the services provided by the `connector` class. In particular, `connector`'s `block` method allows pending requests to be aborted or queued. The `wait` methods block calling reconfiguration threads until ongoing application requests have been serviced using the connector. These primitives are sufficient to allow a broad class of algorithms to be implemented. Once synchronised, meta-level primitives in `component` and `connector` are used to carry out the required structural changes.

For observing algorithm behaviour, OpenRec provides an API for two kinds of monitoring:

- *System monitoring*. Through the `connectionPlugin` interface, plugins can receive events relating to configuration-level changes. In particular, these plugins are notified of structural changes such as components being added, removed, or replaced, connector blocking and unblocking, and changes to a configuration's connectivity.

- *Local monitoring*. Implementations of the `interactionPlugin` interface are registered with particular components and thus receive events from a specified subset of the configuration. `interactionPlugins` also receive events of finer granularity that include details of method invocations.

OpenRec's plugins are intended to be used by instrumentation and visualisation tools which calculate and present the costs associated with reconfiguration algorithms. Plugin instances correspond to observers in the Observer design pattern [8] which once registered with OpenRec receive run-time notifications.

3.2 Openness

OpenRec has been designed to offer three distinct forms of openness:

- *Reconfiguration management*. OpenRec's dynamic change management system is itself open, allowing deployment-time configuration and run-time reconfiguration of the Change Driver and Reconfiguration Manager layers.

- *Application components*. Applications have an open implementation which facilitates their dynamic reconfiguration.

- *Plugins*. Developers are free to write their own plugin components and register them with OpenRec.

Each of these forms of openness stem from the uniform application of reflection and componentisation to both the application and underyling change management system.

In terms of open reconfiguration management, and as described in Section 3.1, OpenRec can be configured with an extensible set of reconfiguration algorithms. This enables developers to create new algorithms as new applications emerge with properties not readily suited to existing algorithms.

For example, a common architectural model for multimedia streaming applications comprises a chain of stateless components, connected by media stream connectors, that progressively process the media stream. The first component of the chain governs the source of the media stream and is connected indirectly to the destination by a number of components that perform tasks relating to coding and compression.

To dynamically replace the intermediate processing components, Mitchell [16] describes an algorithm which creates the chain of replacement processing components, breaks the connection between the source and the first processing component of the original chain, and then connects the source to the first component of the replacement chain. At this point, both chains are active in processing the media stream. Since the original chain is no longer being fed from the source, it will exhaust stream data to process. When this happens, the algorithm switches the connection between the destination component and the last processing component of the original chain to the last component of the replacement chain. The algorithm aims to minimise the period between output from the original chain ceasing and the output from the replacement chain emerging. In this way, the algorithm can maintain a level of QoS where users perceive output to be free of jitter.

Mitchell's algorithm is fundamentally different to the more general algorithms, described in Section 2.3, which require that a safe state be reached and maintained for the duration of change. Specifically, it uses redundancy to exploit the stateless property of components and relaxes the accepted definition of safe state. Furthermore, the architectural model described above more generally conforms to the pipe and filter architectural style [21]. Hence, this algorithm can be used to reduce reconfiguration costs below those that would be experienced using a more general algorithm, in cases where a configuration's style is recognised as pipe and filter.

OpenRec's ability to substitute reconfiguration algorithms at run-time is of particular value where systems are composed from several units, each of which may conform to a different architectural style. For large systems, this is often the case [21]. For example a system which comprises units that conform to the pipe and filter style and others that are based on the explicit invocation style would be appropriately reconfigured using separate algorithms that exploit each style's defining characteristics. Using OpenRec, a variation of Mitchell's algorithm could be used to reconfigure a pipe and filter module and subsequently, Chen's algorithm could be used to reconfigure a different module conforming to the explicit invocation style.

In addition to reconfiguration algorithms, the Change Driver and Reconfiguration Manager are also components that can be changed. A simple implementation of the

Change Driver that we have implemented, for example, uses two known change descriptions to periodically toggle an application between two configurations. This is a case of what Oreizy [18] terms a closed adaptation, since the adaptation is hard-coded. A more sophisicated Change Driver implementation might make intelligent adaptation decisions based on a combination of data received from plugin tools and from querying application components using their meta-level interfaces. In this case, the application can be adapted in unforeseen ways.

OpenRec thus allows adaptive applications to not only change their own behaviour, but in addition they can change the mechanisms for controlling change. The conditions which trigger reconfiguration can be changed by making changes to the ChangeDriver layer. To change the way reconfiguration itself is managed, applications can make changes to the components of the Reconfiguration Manager layer.

3.3 Degrees of Separation

Figure 1 reveals two important degrees of separation within our framework:

- *Change Driver / Reconfiguration Manager*. Reconfiguration algorithms are separated from the Change Driver. As described earlier, the Change Driver communicates change requests, using OpenRecML, to the current reconfiguration algorithm component via the Reconfiguration Manager. The effect of this separation is that `algorithm` components need only be concerned with implementing a particular reconfiguration algorithm, and not with when to carry out reconfiguration or how to determine the actual change required. These latter concerns are the responsibility of the Change Driver for adaptive systems; and for system administrators or change programmers in the case of third-party change for highly available systems.

- *Reconfiguration Manager / Application*. The Reconfiguration Manager is independent of any particular application. Using a Reconfiguration Algorithm component it inspects and adapts application components through their meta-level interfaces. The operations provided by these interfaces are sufficient to implement a wide range of reconfiguration algorithms. This separation simplifies development since developers can work at the application level without regard to specific reconfiguration algorithms and vice cersa. Furthermore it promotes reuse of components within these layers.

4 Example

To illustrate OpenRec's support for highly available systems, we describe a simple EPOS (Electronic Point Of Sale) simulation that we have developed using OpenRec. The simulation comprises three kinds of components: EPOS terminals, a collator, and a report generator. A typical configuration involves a number of EPOS components which are connected to the central collator. When a sale is made at an EPOS component, it sends details of the sale to the collator which acts as a persistent repository for sales data. The collator is also connected to a single report generator to which it periodically sends processed data. Each component type is implemented using OpenRec's component model, specifically inheriting from `component`, and all communication is handled by explicit invocation connectors.

The initial configuration constitutes 5 `epos` components, 1 `collator` and 1 `reportGenerator`. The OpenRecML script that describes this configuration is specified as a parameter in the OpenRec deployment tool along with the name of the default reconfiguration algorithm. In this example, we use the algorithm developed as part of our earlier work [23]. This algorithm has been implemented as a strategy component as described in Section 3.1. According to the classifications described in Section 2.3, the algorithm is a static algorithm whose unit of disturbance is the connector.

For component replacement, the algorithm blocks the source end of connectors (in other words required interfaces). Formally, the algorithm determines the interfaces to block and the way in which they should be blocked as follows:

$$
\begin{array}{l}
\mathit{providedBy} : \mathit{INTERFACE} \pfun \mathit{COMPONENT} \\
\mathit{requiredBy} : \mathit{INTERFACE} \pfun \mathit{COMPONENT} \\
\mathit{boundTo} : \mathit{INTERFACE} \pfun \mathit{INTERFACE}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{dom}\,\mathit{providedBy} \cap \mathrm{dom}\,\mathit{requiredBy} = \emptyset \\
\mathrm{dom}\,\mathit{boundTo} \subseteq \mathrm{dom}\,\mathit{requiredBy} \\
\mathrm{ran}\,\mathit{boundTo} \subseteq \mathrm{dom}\,\mathit{providedBy}
\end{array}
$$

$$
\begin{array}{l}
\mathit{replace} : \mathbb{P}\,\mathit{COMPONENT} \\
\mathit{queueBlock} : \mathbb{P}\,\mathit{INTERFACE} = \\
\quad \{ i : \mathit{COMPONENT} \mid i \in \mathit{replace} \bullet \\
\qquad \mathrm{dom}(\mathit{boundTo} \rhd \mathrm{dom}(\mathit{providedBy} \rhd \{i\})) \} \\
\mathit{abortBlock} : \mathbb{P}\,\mathit{INTERFACE} = \\
\quad \{ i : \mathit{COMPONENT} \mid i \in \mathit{replace} \bullet \\
\qquad \mathrm{dom}(\mathit{requiredBy} \rhd \{i\}) \cap \mathrm{dom}\,\mathit{boundTo} \}
\end{array}
$$

Informally, required interfaces that are connected to a component to be replaced are blocked and any invocations sent on those interfaces are queued during reconfiguration. These interfaces constitute members of the *queue-Block* set. Invocations can be queued since the required interfaces are reconnected following reconfiguration, allowing the replacement component to process them. In addi-

```
<?xml version="1.0" encoding="iso-8859-1"?>
<reconfiguration>
   <replace>
      <original>
         <component
            classname="collator"
            modulename="tests.epostest"
            uniquename="collator">
         </component>
      </original>
      <replacement>
         <component
            classname="collatorTwo"
            modulename="tests.epostest"
            uniquename="collator">
         </component>
      </replacement>
   </replace>
</reconfiguration>
```

Figure 4. OpenRecML reconfiguration script

tion to these required interfaces being blocked, the required interfaces of the component to be replaced are also blocked, but this time using an abort block (*abortBlock* set members). In this case, further outgoing invocations are aborted. A component which is to be replaced is synchronised when it has completed processing any incoming requests on its provided interfaces and when any invocations made on its required interfaces, have completed. Queued invocations are not available to the component to be replaced and abort blocking its required interfaces means that all interaction involving the component will terminate in bounded time.

Reconfiguration is initiated by submitting an OpenRecML reconfiguration script to the Change Driver. The script describing `collator`'s replacement is shown in Figure 4. The script simply identifies the component to replace using its unique name in addition to the component type to instantiate as its replacement. Binding details are not required in this case since the algorithm requires that the replacement component provides at least those interfaces of the original component. The algorithm's `doCheckConstraints` method includes a check to verify this using `component`'s meta interfaces to discover the provided interfaces of the two components.

Figure 5 shows a simple visualisation tool which renders components and their connectors. Directed links represent connectors and point from required to provided interfaces. The tool is an OpenRec plugin and implements the `connectionPlugin` interface which, as described in Section 3.1, means that it receives system-wide events describing configuration changes. The tool records the configuration events and enables users to traverse, back and forth, through configuration changes over a system's lifetime.

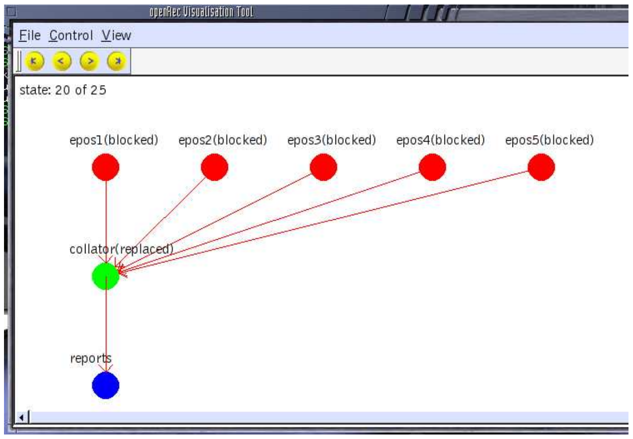
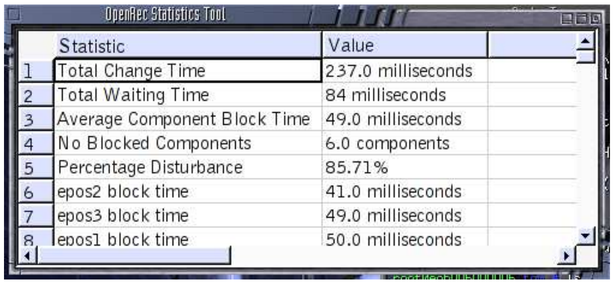


Figure 5. Visualisation tool

Figure 5 shows that at state 20, the application has synchronised with change, since all `epos` components have (queue) blocked on their required interfaces, and that the `collator` has been replaced. Prior to this state, `collator`'s required interface would have been blocked in abort mode, preventing it from making further requests of `reportGenerator`. In addition, since `collator`'s encapsulated state should survive reconfiguration (it is cumulative sales data), the algorithm extracts the state from the original `collator` and uses it to initialise the replacement. The algorithm does this using a state management interface that both the original and replacement `collator` components are required to provide. Ensuring that the components do provide the necessary interface is verified as part of the algorithm's `doCheckConstraints` method using the `metaData` class. In states 21 onwards, blocked interfaces are unblocked allowing the components to resume normal execution.

In addition to the visualisation plugin, we have also implemented a basic instrumentation plugin which presents quantifiable reconfiguration data. Similarly to the visualisation tool, the statistics tool shown in Figure 6 implements the `connectionPlugin` interface. The statistics tool displays the total reconfiguration time measured from the time reconfiguration is initiated to the time normal execution of all components is resumed. The total wait time is part of the synchronisation period where the algorithm waits for application components to process any ongoing requests necessary for the application to synchronise. The remainder of the reconfiguration time is spent determining the members of the *queueBlock* and *abortBlock* sets and actually carrying out the structural changes. In this simple configuration, application disturbance is high since all 5 `epos` components and `collator` experience some form of blocking.


OpenRec Statistics Tool

	Statistic	Value
1	Total Change Time	237.0 milliseconds
2	Total Waiting Time	84 milliseconds
3	Average Component Block Time	49.0 milliseconds
4	No Blocked Components	6.0 components
5	Percentage Disturbance	85.71%
6	epos2 block time	41.0 milliseconds
7	epos3 block time	49.0 milliseconds
8	epos1 block time	50.0 milliseconds

Figure 6. Reconfiguration measurement tool

The statistic tool's output is useful in infering the impact reconfiguration has on an application's non-functional requirements. For example, of all the `epos` components, `epos1` experiences the longest blocking time of its required interface. This delays the response time for `epos1`'s request by up to 50 milliseconds. For this system, such a small delay is unlikely to violate its response time requirement. However, for more realistic implementations with longer processing times, response times experienced by users may drop below tolerable levels during periods of reconfiguration. Moreover, for other applications with more stringent timing constraints, such as the multimedia streaming system discussed earlier, knowledge of the reconfiguration overhead becomes increasingly more important. In cases where constraints are not met, developers can use OpenRec to experiment with alternative algorithms and configuration structures.

5 Related work

Reconfigurable distributed object systems [20] [1] [4] and other component-based systems [15] [23] are restricted by closed implementations. In particular, these systems encapsulate a single fixed reconfiguration algorithm. In addition to being closed, these systems do not offer support for measuring the costs associated with dynamic reconfiguration. In these respects, they are in contrast to OpenRec.

The ArchStudio tool suite [19] has been designed to reconfigure components-based systems that conform to the C2 architectural syle. This style prescribes a hierarchy of components that interact using only asynchronous messages and events. C2 applications are open in that developers may encode a range of reconfiguration policies in connectors. However, in comparison with OpenRec, this enables only a limited form of openness and blurs the separation of reconfiguration and application concerns. In addition, ArchStudio mandates use of the C2 style whereas OpenRec is independent of any particular style.

Self-healing systems [5] monitor themselves and initiate self-repairing action when a constraint fails to hold true. Recent work, including [2], [10], [7] and [18], addresses the problems of expressing architectural-level constraints (throughput and response time for example), monitoring systems at run-time, mapping monitored data to architectural constraints, and triggering repair actions. However, they assume the existence of lower-level algorithms to handle synchronisation and persistent state management. While such work leads to valid reconfiguration at the architectural level, applications may fail because of unmanaged reconfiguration at the component level.

6 Conclusions and further work

In this paper, we have presented an overview of OpenRec, an open framework for managing dynamic reconfiguration. We have demonstrated that OpenRec offers three novel features. First, OpenRec accommodates an extensible set of reconfiguration algorithms. This addresses the issue that one algorithm does not fit all applications. Second, using OpenRec's extensible plugins, developers can measure the costs incurred in using a particular reconfiguration algorithm. This is useful in evaluating alternative application designs for suitability for reconfiguration. Furthermore, based on a comparative analysis of multiple algorithms, an informed decision can be made as to the most appropriate algorithm for managing a particular scenario. Third, OpenRec is itself reconfigurable. In particular, reconfiguration algorithms can be substituted at run-time, as can other key elements of the OpenRec architecture. These features are facilitated by the use of OpenRec's underlying reflective component model.

The current status of our work includes a Python implementation of the reflective component model, two static reconfiguration algorithms, a basic Change Driver, and the two plugin tools described in Section 4. While this has enabled us to verify the key novel contributions outlined above, there remains much interesting further work which we expect to report on in the near future. In particular:

- *Comparative studies of reconfiguration algorithms.* We expect to further validate OpenRec's support for implementing reconfiguration algorithm and its standard form for capturing algorithms' constraints. In particular we intend to implement dynamic algorithms in addition to more specialist algorithms that are style-specific. The former require run-time information that is available using OpenRec's meta-level interfaces. From the growing catalogue of algorithms, we will collect their relative costs and hope to formulate heuristics governing their use.

- *Deeper analysis of reconfiguration.* The time taken to carry out reconfiguration is dependent on what the application is doing when reconfiguration is initiated.

For example, performing the reconfiguration described in Section 4 yields different figures for disturbance times each time it is run. Thus, best-, average-, and worst-case times could be generated by taking a sufficient number of samples. Beyond this, more complex analysis would allow a reconfiguration request to be deferred to the time it impacts least on the running application. We anticipate that OpenRec's monitoring capabilities can be developed to support this level of analysis.

- *Briding the gap between component and architectural based change management*. The research that others are doing at the architectural level, in terms of adaptation to meet an application's non-functional requirements, is complementary to our lower-level change management. Integrating these approaches would enable a system to be monitored, the need for and the type of adaptation to be determined and then carried out efficiently using an appropriate algorithm that preserves system integrity.

- *Self adapting change management*. Finally we would like to investigate the possiblity of developing a Change Driver that can reason about the system to be adapted using data obtained via plugin tools and meta-level interfaces to automatically select the most appropriate algorithm for handling adaptation. More generally, we intend to explore OpenRec's inherent openness for reconfiguring its reconfiguration management capabilities.

References

[1] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. In *4th International Conference on Configurable Distributed Systems, pages 35–42. IEEE Computer Society Press*, 1998.

[2] G. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas. Reflection, self-awareness and self-healing in openorb. In *Workshop on self-healing systems*, 2002.

[3] G. S. Blair, G. Coulson, A. Anderson, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski. The design and implementation of open orb 2. In *Distributed Systems Online, Vol. 2. No. 6*, 2001.

[4] X. Chen and M. Simmons. Extending RMI to support dynamic reconfiguration of distributed systems. In *22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, 2002.

[5] S. Cheng, D. Garlan, B. Schmerl, J. P. Sousa, B. Spitznagel, and P. Steenkiste. Using architectural style as a basis for system self-repair. Technical report, Carnegie Mellon University, 2002.

[6] M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In *IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.

[7] E. Dashofy, A. Hoek, and R. Taylor. Towards architecure-based self-healing systems. In *Workshop on self-healing systems*, 2002.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Elements of reusable software. *Addison Wesley*, 1995.

[9] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. In *IBM Systems Journal, Automonic Computing. Vol 42. No 1.*, 2003.

[10] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *Workshop on self-healing systems*, 2002.

[11] K. M. Goudarzi. *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*. PhD thesis, Imperial College, London, 1999.

[12] G. Kiczales, J. Rivieres, and D. Dobrow. The art of the meta-object protocol. In *MIT Press*, 1991.

[13] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[14] M. M. Lehman and L. A. Belady. Program evolution. *Academic Press, APIC Studies in Data Processing No 27*, 1985.

[15] J. Magee, J. Kramer, and M. Sloman. Constructing distributed systems in conic. *IEEE Transactions on Software Engineering*, 15(6), 1989.

[16] S. R. Mitchell. *Dynamic Configuration of Distributed Multimedia Components*. PhD thesis, University of London, 2000.

[17] M.Roman, F.Kon, and R.H.Campbell. Reflective middleware: from the desk to your hand. In *IEEE Distributed Systems Online, Special issue on Reflective Middleware, Vol. 2, No. 5*, 2001.

[18] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. In *IEEE Intelligent Systems, Vol. 14 no. 3, pages 54-62*, 1999.

[19] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *International Conference on Software Engineering, Kyoto, Japan*, 1998.

[20] J. Paula, A. Almeida, M. Wegdam, M. van Sinderen, and L. Nieuwenhuis. Transparent dynamic reconfiguration for CORBA. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications*, 2001.

[21] M. Shaw and D. Garlan. Software architecture: Perspectives on an emerging discipline. *Prentice-Hall, Englewood Cliffs, NJ*, 1996.

[22] C. Szyperski. Component software: Beyond object-oriented programming. *Addison Wesley*, 1998.

[23] I. Warren. *A Model for Dynamic Configuration which Preserves Application Integrity*. PhD thesis, Lancaster University, 2000.

[24] M. Wermelinger. A hierarchic architecture model for dynamic reconfiguration. In *2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE '97)*, 1997.