Dynamic Adaptation of Dependable Systems

Jamie Hillman & Ian Warren
Computing Department, Lancaster University, Lancaster, UK
Email: {j.hillman,iw}@comp.lancs.ac.uk
Thesis submission date (Jamie Hillman): 2004

Abstract

Dependable software systems, like any other software system, are subject to change during their lifetimes. Traditional approaches to bringing about change require that the system be brought offline temporarily. However, this is often undesirable due to requirements for high availability. Furthermore, dramatically reducing availability may also cause a reduction in other dependability attributes such as security, safety and reliability. This paper presents an overview of a framework for managing change in dependable systems dynamically with the aim of preserving a high degree of availability. The framework allows the dynamic change process, controlled by an open set of algorithms, to be monitored and the impact of particular algorithms on a system’s dependability attributes to be revealed. Novel aspects of our work include support for making an informed decision about the algorithm used to control the change process and the means to assess the dependability of change control algorithms.

1 Introduction

Dependability is often the most important characteristic of a system providing a critical service. While many of a systems attributes contribute to it’s dependability, four are principal[3]:

Availability. The likelihood of the system being available to service a request at any time.

Security. The ability of the system to keep confidential information undisclosed, to disallow improper alterations to information and to preserve availability in the face of attack.

Safety. The non-occurrence of dangerous consequences on the environment of the system.

Reliability. The probability of failure in servicing a request.

The change process is largely overlooked and creates a high cost of change in traditional dependable systems. Change is usually introduced statically by taking the system offline and introducing the changes, then restoring the system with the new implementation in place. This approach causes a total loss of availability for a period of time and can affect other dependability attributes. For example if a system which controls door locks was taken offline the doors may all be locked (fail secure), causing a loss of safety as people may be trapped, or all unlocked (fails safe), causing a loss of security. There are many ways in which one dependability attribute can be affected as a consequence of a change in another attribute. For example if a system’s security is compromised it could be vulnerable to attack which in turn could cause it to fail, thus decreasing its availability. As static change dramatically affects at least one attribute it is obviously not a favourable choice for bringing about change. This leads us to examine ways in which change can be introduced dynamically.

Dynamic change management involves bringing changes into effect whilst the system is functioning. This is preferable because it means that the system isn’t taken offline completely and so availability is kept at a maximum, as shown in figure 1.

With component based systems, change can be carried out by replacing components and manipulating the connections between those components. Take for example a twenty-four hour super-market system which consists of ten electronic point of sale (EPOS) components con-
2 Dynamic Adaptation

Various dynamic reconfiguration algorithms have been developed and each has its own characteristics and applicability constraints. These characteristics and constraints will affect the dependability characteristics of the dynamic adaptation process. This section identifies some key differences between algorithms and the effects they have on a system’s dependability attributes.

Algorithms can be split into two categories governing how they reach a safe state where change is synchronised with a running system. The first category of algorithms are deterministic and select the components or connections to block based on the connections that exist between components involved in change. Returning to the EPOS example, all components with a connection to the data collector had their connections blocked.

Non-deterministic algorithms only block those components that actually attempt to start communicating with a component to be operated on. This means that there are potentially fewer blockings as the condition for blocking is more restrictive than that of the simpler deterministic algorithms.

For example applying our own algorithm [5] to the shopping system described above, the connectors connecting the EPOS components to the data collector would be blocked first so that no further transactions could be initiated by the EPOS components upon the data collector. Instead invocations on this connector would be buffered for the duration of the change. After waiting for all ongoing transactions to complete the component can safely be removed and replaced with a new implementation, and the connections re-established and unblocked. In this situation there is no disruption to transactions and the EPOS components have only to wait for a short period of time before the replacement component is available to service their requests.

Whilst the EPOS example is a simple one it illustrates the need for a managed dynamic change process. Furthermore, the example shows that although there is a short loss of availability of the data collector component, components that are independent of this remain unaffected and available. As the graph in Figure 1 shows, this is in clear contrast to the static approach.

Dynamic reconfiguration algorithms have emerged which aim to preserve application integrity during periods of dynamic change. Change operations such as add, remove and replace operate on components and connectors and their application is controlled by such algorithms. The fundamental role of an algorithm is to synchronise change with a running system which typically involves guiding the system toward a safe state by blocking certain components and/or connections.

Figure 1: Availability over time with static and dynamic change

connected to one data collector component. If the data collector is to be upgraded then the component could be disconnected and replaced with a new implementation. The new implementation would then be reconnected to the EPOS components.

The changes carried out in the above scenario are likely to cause problems if not carried out in a controlled manner. This is because any transaction taking place between the EPOS components and the data collector at the time of removing the data collector would be interrupted. This could leave the system in a state where the EPOS component believes the transaction completed successfully when it did not, causing inconsistencies. There are various approaches to solving this problem that rely on embedding change management functionality in component implementations but it would be preferable if components could be added, removed and replaced more gracefully and in an application independent manner. Alternatively there is the static approach but this would involve taking the EPOS components offline, meaning a costly loss of availability.

Dynamic reconfiguration algorithms have emerged which aim to preserve application integrity during periods of dynamic change. Change operations such as add, remove and replace operate on components and connectors and their application is controlled by such algorithms. The fundamental role of an algorithm is to synchronise change with a running system which typically involves guiding the system toward a safe state by blocking certain components and/or connections.
When availability is absolutely paramount nondeterministic algorithms may be more preferable as they will block only those parts of the system absolutely necessary for safe change. On the other hand, deterministic algorithms may be preferable when the time in which the change process completes is important, as they spend less time determining which components or connections must be blocked.

Some algorithms block whole components [2] whereas others only block selected connections [5]. The latter may preserve a higher degree of availability, but again at the cost of additional complexity and run-time overhead in determining what is to be blocked.

A further algorithm [1] distinguishes between those invocations that modify a component’s state and those that do not. Essentially, the algorithm blocks further mutator requests and waits for any ongoing mutator requests to complete so that the state of the component is not left inconsistent. Having reached this safe state, any ongoing selector requests are aborted and must be resent following reconfiguration. The effect of this behaviour is to minimise the synchronisation time, but the component may appear unavailable for longer, when compared to other algorithms, to components that invoke its selector operations. The system availability profile for this algorithm will thus be different to that of other algorithms.

Finally there are characteristics not related to synchronisation which may affect dependability attributes. For example if an algorithm discards critical state when replacing a component it could cause a dramatic reduction in reliability and perhaps safety in some cases. For example, in the EPOS scenario, discarding the state of the data collector would result in the system being unreliable since following reconfiguration it would have lost knowledge of completed transactions. If state is not important, however, it need not be handled during change. For example a lookup service component may cache the results of recent lookups. If the state is lost in replacing that component it will only lead to slower lookups. This may be considered worthwhile in light of the increased speed of replacement and so greater availability.

So far it has been shown that different algorithms behave in different ways and that this can affect dependability attributes in various ways, often leading to trade-offs in different attributes. The conclusion here is that one algorithm is not optimal for handling the specific dependability requirements for different systems. Consequently, it is desirable to have support for examining the impact on dependability that different algorithms exhibit as they manage dynamic change.

Existing systems such as [4][5][2] contain reconfiguration algorithms but they are fixed and closed. Hence with these systems, it is not possible to adapt the algorithm to the specific dependability needs of a system or to substitute the algorithm with a completely different one. We argue that it should be possible to examine the effects of an algorithm at run time, to substitute algorithms and to implement new algorithms. If these requirements are satisfied, the most appropriate algorithm can be chosen or developed for a system to match its dependability requirements. Furthermore, the behaviour of the algorithm and the resulting dependability attributes should be measurable and certifiable.

3 Our Framework

Our framework for dynamic change management is implemented upon a reflective component model and is suitable for hosting component based applications. Figure 2 introduces the architecture of the framework and shows that it is split into three conceptual layers: Change Driver, Adaptation Manager, and Application. The Change Driver determines if and when a change is necessary. To initiate
change, it generates a description of the change, expressed in xADL, and passes the description to the Adaptation Manager, which is responsible for adapting the running application using a particular reconfiguration algorithm. The application is implemented in the Application layer using our reflective component model, allowing reconfiguration algorithms to manipulate application components using their meta-level interfaces.

3.1 Support for comparing reconfiguration algorithms

Using our framework, a developer is able to build a dependable adaptable system without being concerned with the characteristics and constraints of particular reconfiguration algorithms. The framework allows developers to configure the Adaptation Manager with one of several reconfiguration algorithms. In addition, developers are able to observe the behaviour of different algorithms and consequently the costs and trade-offs in dependability attributes associated with using them. Adaptation costs are measured in terms of the number of components effected by an algorithm, the way in which components are effected, the duration of these effects and the total time required to make the change.

Each reconfiguration algorithm is implemented as a component using our reflective component model. To allow the Adaptation Manager to be configured with one of many algorithms, we have used the Strategy design pattern where each algorithm corresponds to a concrete strategy. In response to an adaptation request, the Adaptation Manager delegates the request along with the xADL description to the current Reconfiguration Algorithm component. This component may respond initially by optimising the xADL description by, for example, changing the order of instructions or arranging instructions to be performed concurrently. Regardless of any optimisation, the Reconfiguration Algorithm component proceeds by determining whether its constraints are satisfied by application components. To do this, the component uses the meta-level interfaces in the Application layer. Application components and connectors have tuples associated with their meta-level interfaces that allow properties to be set and inspected. An example component property is whether the component implements a particular change management interface, such as that required by Kramer and Magee’s algorithm. Other properties might indicate whether a component encapsulates state. Where the set of components involved in an adaptation satisfy the algorithm’s constraints, the Adaptation Manager allows the Reconfiguration Algorithm component to carry out the reconfiguration, using the Application layer meta-level interfaces.

For observing algorithm behaviour for the purposes of dependability certification and assessing the suitability of an algorithm for managing a particular run-time change for some system, the framework provides an API for plugins at two points:

- **Adaptation Manager layer** Plugins at this level receive events regarding the progress of the Reconfiguration Algorithm component when reconfiguring an application. Particular events generated by this layer include those which inform registered plugins about any optimisations made by the algorithm and progress events describing which instructions have been executed.

- **Application layer** At this level, plugins receive events that describe changes in terms of components and connectors during the adaptation process. These events are particularly useful for tools to measure the costs, introduced above, associated with the current algorithm. In general, plugins are intended to implement monitoring tools. Plugin components conform to observers that register interest in the Adaptation Manager or the Application layer subjects, according to the Observer design pattern.

3.2 Openness

The framework has been designed to offer three distinct forms of openness based on the uniform use of components:

- **Reconfiguration algorithms** As described above, and unlike existing change management systems, the framework can be configured to use one of many reconfiguration algorithms to manage adaptation. Furthermore, the set of reconfiguration algorithms is extensible. Developers can use an API to encode additional Reconfiguration Algorithm components as new kinds of applications emerge with properties not readily suited to existing algorithms. Algorithms can also be substituted at run-time, allowing one algorithm to manage one adaptation and then for a different algorithm to manage a subsequent
adaptation. This is particularly useful where different parts of a system have different dependability requirements and thus different algorithms are appropriate for reconfiguring them.

Plugins Developers are free to write their own plugin components at the Application and Adaptation Manager layers. Plugins typically include instrumentation, analysis and visualisation tools for the purpose of studying the behaviour of algorithms.

Core components The Change Driver and Adaptation Manager are themselves components that can be changed. A simple implementation of the Change Driver, for example, could use two known change descriptions to periodically toggle an application between two configurations. This is a case of what Oreizy [4] terms a closed adaptation, since the adaptation is hard-coded. A more sophisticated 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.

4 Conclusions and further work

In this paper, we have shown that dynamic change preferable to static change where availability and other dependability attributes of a system should be preserved. Dynamic change can be a dangerous process when not controlled by dynamic reconfiguration algorithms. A multitude of such algorithms have emerged, each of which affects a system’s dependability attributes in different ways. We have presented a framework which reveals these differences. The current status of the implementation is a complete reflective component model which has been used to implement a simple Change Driver, one Reconfiguration Algorithm component, an Adaptation Manager and basic monitoring plugins.

We have argued that one dynamic reconfiguration algorithm is not suitable for all systems and so we have designed the framework to be open and flexible, allowing change in the change management system itself. The framework also allows new and existing algorithms to be implemented in a standardised form and compared and analysed through the use of plugins. Openness and clean separation of concepts are central to our approach and essential in a adaptation framework that is to deal with many change control strategies, change drivers and applications. We believe that these design features make our framework an enabler for broad classes of dependable and adaptable systems.

Our plans for future work include:

Monitoring tool support for dependability certification We aim to develop statistical analysis and visualisation plugins that provide sufficient information on the behaviour of algorithms to assess the suitability of an algorithm to a given situation and to certify an algorithm to a given level of dependability.

Self-adapting change management Finally we would like to investigate the possibility of implementing a change driver that can reason about the system to be adapted. This change driver could have knowledge of the algorithms available in order to make sure that the dependability levels required are met by selecting the appropriate algorithm automatically.

References


