An Application Directed Adaptive Framework for Autonomic Software

Bridget Meehan
Intelligent Systems Research Centre, University of Ulster, Derry, Northern Ireland
Email: ba.meehan@ulster.ac.uk

Girijesh Prasad and T.M. McGinnity
Intelligent Systems Research Centre, University of Ulster, Derry, Northern Ireland
Email: {tm.mcginnity, g.prasad}@ulster.ac.uk

***Abstract*—Autonomic computing is gradually becoming accepted as a viable approach to achieving self-management in systems and networks, with the goal of lessening the impact of the complexity crisis on the computing industry. The authors propose the integration of high level self-organisation features into an Application Directed Adaptive Framework (ADAF), an autonomic-oriented software development process, which when used during the development of software applications, enables those applications to exhibit autonomic behaviour. This paper discusses the infrastructure of the ADAF and demonstrates two self-managing capabilities that come about in a software application as a result of applying the ADAF, namely self-monitoring and self-diagnosis.**

***Index Terms*—autonomic computing, autonomic-oriented software development process, self-organising features, self-monitoring, self-diagnosis**

I. INTRODUCTION AND BACKGROUND

The computing industry has a tradition of concentrating on building smaller, faster, cheaper machines. As a consequence, the last two decades have seen a steady decline in the cost of computer hardware and a dramatic increase in processing capabilities, storage capacities and communication speeds. This race towards 'smaller, faster, cheaper' has resulted in the ubiquitous use of independent, heterogeneous devices, applications, and systems that is so pervasive and far-reaching that most aspects of our everyday lives have been touched or altered irrevocably.

Software has had to change too, to keep up with the pace of hardware changes and the variety of needs and expectations of increasingly more sophisticated and demanding users [1, 2]. The days of command-line interfaces with black displays and a pulsing cursor in the top left hand corner of the screen are long gone, given way to intuitive, graphical user interfaces with multi-modal forms of interaction and menus of elaborate functionality.

However, the pursuit of the 'smaller, faster, cheaper' goal has not come without its consequences and, the complexity arising from the billions of interconnections between heterogeneous devices and systems and users is growing beyond human ability to manage [3, 4, 5]. This phenomenon is labelled the complexity crisis, a crisis that threatens to undermine the benefits proffered by technology and hinder further progress [6, 7]. Indeed, the complexity crisis is already impacting on the computing industry in terms of escalating administration and financial costs, reduced reliability and integrity, as well as software specific impacts such as increased development and maintenance effort and reduced user confidence [8, 9].

There is consensus that if the computing industry is to manage complexity, there must be a profound change in how technologies are constructed, a change that moves towards developing technologies that can manage themselves (and thus the complexity) without the need for vast numbers of expensive human administrators [4, 10].

Software must also be steered towards greater autonomicity, with a shift away from the development of rigid and often brittle programs that are heavily dependent on human intervention, and unable to adapt or evolve to meet unforeseen requirements or conditions. Instead, there is a need for software that possesses the capability of dynamic evolution; that allows bug fixes and upgrades to be integrated without having to stop application execution; that detects potential problems and errors and takes action to prevent or recover from them; that restructures itself by accommodating multiple design choices and dynamically selecting from among them [11-14].

Autonomic computing is a biologically-inspired approach focused on reliably and robustly dealing with complexity and uncertainty in technology and software. Autonomic computing systems should aim to be capable of self-managing with a minimum of human interference in order to provide reliable, always available, robust services [8, 15]. Autonomic systems should be capable of anticipating changing requirements and conditions, of adapting to those anticipated changes by reconfiguring and optimising system structures, of protecting themselves from security breaches, of repairing

themselves when errors and failures occur, and all without interrupting execution and with little or no human assistance [1, 16].

Four key self-managing attributes are identified in autonomic computing, specifically self-configuration, self-optimisation, self-repair, and self-protection. These basic four have been extended to include a wide range of what is called self-* (pronounced *self-star*) attributes which describe any activity that can be performed by a system without the need for complete human intervention [17, 18]. Some additional self-* attributes include self-evolution, self-diagnosis and self-monitoring.

A large proportion of the research in autonomic computing focuses on autonomic capabilities within the context of distributed networks and systems and web and server applications, with less attention given to more stand-alone software applications which are not necessarily distributed or available over the Internet or other networks, and which are composed of components with relatively low levels of granularity such as functions, objects, or object methods. For this research, the focus is on such software applications and how they might be developed for autonomicity.

A. An Approach to Realising Autonomic Software

In exploring ways in which autonomic software applications might be realised, it was observed that many of the aims of autonomic computing are compatible with the features of self-organising systems [6, 7, 19, 20]. Although self-organisation is an extensive area of research in its own right, some of the high level features from self-organising theory can be simplified and extracted for the purposes of creating a framework for autonomicity.

Self-organisation is a process whereby global order or structure emerges from an entity comprised of a collection of autonomous and disordered, but interconnected, components. Components act based on local knowledge to achieve a simple task and collectively their interacting behaviours emerge as more complex and ordered higher level (global) behaviours that create order and structure in the entity. The process happens without the supervision of a 'leader' or of a controlling component and without any external pressure or constraints.

Based on this, a self-organising system can be defined as one that is composed of autonomous components, each of them acting independently of each other without higher supervision, and each following a few simple rules based on local knowledge that achieve organised behaviour and order [5, 19, 20]. Examples of self-organising systems include insect colonies (bees, ants, and termites), the human brain and liver, swarms and flocks, traffic jams, markets, economic systems, ecosystems, and societies [5, 20, 21].

There are a number of features common to self-organising systems, and although not all of them are entirely feasible or even necessary in a software application, some of them can be usefully extracted, to greater or lesser degrees, to aid in the creation of autonomic software. For instance, self-organising systems are able to adapt to unforeseen changes, problems and events in their environment, and can re-configure their structure to fit the conditions demanded by the environment at that time.

A self-organising system forms without the need for any central or external controller and instead, control is distributed evenly over the whole of the system, with all components contributing equally to the emergent order and structure. The distributed nature of self-organising systems means that they are inherently robust and fault-tolerant, and can withstand errors, disruption or partial destruction because non-damaged regions can usually make up for the damaged ones. It also means they are capable of restoring themselves and (self-) repairing any damage caused.

Self-organising systems result in emergent behaviour that results from the autonomous components acting on simple rules. It is these features that the authors have sought to implement (entirely or in part) in the ADAF, with the aim of enabling applications, developed using the ADAF, to exhibit autonomic behaviour.

A preliminary vision of this work was presented in the conference paper [22]. The current paper is organised in the following way. Section two provides a brief summary of work related to this research. Section three discusses the design and implementation of ADAF and section four follows on from this by detailing a case study in which ADAF was applied and tested. Finally, section five ends with a summary.

II. RELATED WORKS

Investigating autonomic computing from the point of view of architecture-based adaptation and evolution is related to the research in this paper, and is classified under a range of names, such as dynamic software architectures, runtime evolution, adaptive dynamism, self-organising systems, intelligent dynamism, self-repairing systems and self-adaptive software. With this approach, systems are described and modelled at the architectural level. The architecture of a system is an abstraction of its structure and behaviour, described as a set of connected components, their visible properties and the relationships or bindings between them. Examples can be found in [23-26].

Research into architecture-based adaptation focuses on the large-scale, distributed, and component-based or agent-based systems used in Internet, middleware, mobile and client-server environments, and ad hoc and peer-to-peer networks. Systems of this nature typically contain coarse-grained components such as groups of collaborating objects, agents, or entire applications. There is no evidence to demonstrate where this approach has been applied to the type of applications that are the focus of this research.

The area of adaptation and dynamic updating is hugely significant to autonomic computing. Dynamic adaptation or updating is where code has facilities for selecting and incorporating new behaviours at runtime [27]. Until recently, few commercial systems have required changes to be made on the fly and suitable technologies to

facilitate this sort of change have not been rigorously developed, although this is changing as the recognition for the need for dynamic updating grows.

This area uses techniques such as proxies, partial system shut-down, and dynamic linking and loading [28]. For instance, the Fifi architecture focuses on evolving Java programs. In [29], dynamic patches containing both the updated code and the code needed to transition from the old version to the new are applied to a running program. Dynamic updating for C++ classes is investigated by [30, 31]. A C++ proxy-based approach is applied to dynamic updating in the work of [32].

Research in this area has some focus on non-distributed, fine-grained applications which are the focus of this research. They also employ techniques such as dynamic linking, proxies, and redirection code, as does this research. However, unlike this research, these approaches do not have a facility for change management, and often they are incapable of reasoning about, specifying, or controlling changes, or of selecting between versions of classes. As such, these approaches do not encompass the wider issues surrounding autonomic computing, and they do not focus on the use of dynamic updating for the purposes of autonomicity which this research does.

Agent-based technology is closely related to the notion of self-organising and autonomic systems since agents are essentially autonomous entities. In this approach, applications and components are written as software agents that communicate with each other by sending and receiving messages using an Agent Communication Language. Agents are active entities that have their own thread of control which extends over both state and behaviour. This means they are reactive, that is, they respond to changes in their environment; and they are proactive, that is, they adapt goals and take the initiative.

While agent-based technology is certainly a viable approach to autonomic systems, many agent systems use interpreted languages which are not efficient enough for low-level processing. Additionally, due to the communication overhead in current agent systems, an application is often implemented as a few large agents rather than many small agents, resulting in large replacement units that are too coarse-grained for many applications, including those that are the focus for this research. Agent-based technology also requires an investment in specialised software and retraining for engineers, while the research in this thesis uses technologies and techniques that are widely used in commercial programming [32, 33, 34].

III. THE APPLICATION DIRECTED ADAPTIVE FRAMEWORK

The ADAF is a proof-of-concept autonomic-oriented software development process. Realised in software, it embodies a number of capabilities that reflect a number of selected self-organising features. The ADAF distributes the control of management and coordination activities as much as possible and operates largely at a component level. Where distributed control might interfere with the achievement of global needs and goals, control is central and operates at a global level.

The ADAF has the ability to self-adapt. This means it can evolve and reconfigure, and therefore can support multiple versions or states of software, as well as dynamically add to and select between them, without requiring the application to stop execution. The ADAF has monitoring and reasoning capabilities so that behaviour can be observed and reasoned about, thus making it possible to select between versions. The ADAF has the ability to self-repair. This requires that behaviour can be monitored and reasoned about to determine if there are any problems. The ADAF also allows application components to have simple, localised rules that they can act upon.

The ADAF was developed and deployed on standard hardware: a Toshiba Tablet Personal Computer with 512MB of RAM and an Intel Pentium Processor with a clock speed of 1.80GHz, running on the Windows XP Tablet PC operating system. It was developed in the Microsoft Visual Studio .NET Framework, Version 1.1 and written in Microsoft Visual Studio .NET C++. The ADAF did not require modification of the C++ language but was able to use the mechanisms already available.

A. ADAF Components

The component parts of the ADAF are cells. ADAF cells are of two types: local and global. All cells are comprised of a complex of software programs and database files. When the ADAF is used in the development of a software application, it enables that application to achieve autonomicity, and the end product is an autonomic application that is ADAF-embedded. An ADAF-embedded application is one where each module, for example each object, of the application has a local cell attached to it, and where the overall application has a global cell embedded into it for the purposes of global monitoring and reasoning, and application evolution. An ADAF-embedded (object-oriented) application is depicted in Fig. 1.

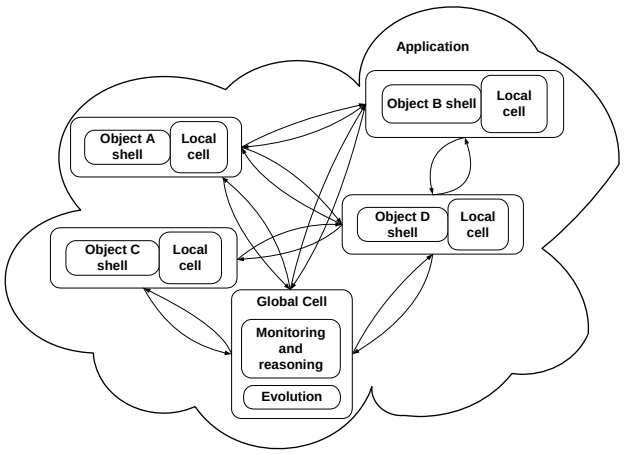

Figure 1. An ADAF-embedded application, including the global cell.

B. ADAF Local Cell Structure

There is an ADAF local cell for each object in the application. When a local cell is embedded into an object, the object itself becomes a shell used only as a conduit between the object's local cell and the object's external environment. Fig. 2 provides a logical conceptualisation of a local cell embedded in a specific object, object A. Object A is an exemplar only and represents any object. The local cell for object A operates as a control loop that contains four interacting elements: local monitoring; local reasoning; stored rules for selecting object code; and a library of object code.

The loop works in the following way: the stored rules represent the conditions under which object code executes; the behaviour of the application is monitored by collecting runtime data; the monitored data are compared against the stored rules to reason about the object code from the library that should be executed; the behaviour of the library code executed is monitored, thus closing the loop.

In the local cell, object code is kept separate from local cell code. This ensures a loose coupling between the two types of code, otherwise known as a separation of concerns. There is also a separation in the objects themselves, between the object definition and object code, and thus object A becomes a shell containing the definition of the object's attributes and methods, while the implementation of the methods is extracted and placed in the library of object code. The methods that remain in the object shell need only provide code that redirects execution to the local cell. The separation of the object code from the object definition into a library is further necessary as a means of allowing multiple versions of code to exist and of allowing new or updated versions of code to be added.

C. ADAF Local Cell Library of Object Code

The library of object code is realised using dynamic link libraries (DLLs). DLLs are units of code capable of being loaded into an application at runtime and executed, that is, dynamic loading. Since the level of change is directed at the method level, it is the case that each object method is separated into its own DLL. Therefore, when a local cell is initially embedded in an object, there is one DLL in the library for every object method.

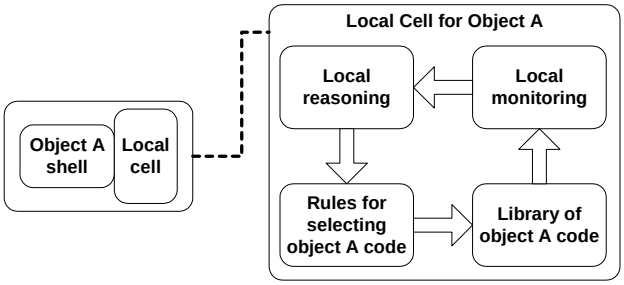

Figure 2. Logical conceptualisation of a local cell embedded in object A.

As such, the library can be described as a store of DLLs. Throughout the lifetime of the application, as needs and goals change, the library can dynamically change as required, allowing multiple versions of DLLs to be added, existing DLLs to be replaced, and entirely new DLLs to be added.

D. ADAF Local Cell Rules for Code Selection

That object code resides in the library in the first instance, and that multiple versions of code can exist as well, means that there must be a mechanism for selecting between DLLs. Rules provide the means for selecting between DLLs of object code. They specify the conditions that must be met before a DLL can be executed. These rules are defined as parameter and threshold values. Thresholds represent a baseline or acceptable level of application performance and behaviour and can be a single value or a range of values. Thresholds are derived from application requirements and objectives, and also from the experience of developers and designers.

During application execution, meeting or exceeding threshold ranges or values helps determine which methods are selected for execution. Thresholds can change over time (that is, they are refined) as knowledge of application usage and performance grows. A parameter value represents a specific value that an application variable is expected to have at runtime. These values are usually derived from application requirements and objectives. During application execution, when an application variable matches a parameter value, it helps determine which DLLs are selected for execution.

The rules can serve local or global application needs and goals. Global rules can apply to a single local cell or over multiple local cells and, are based on application needs and goals, for example, if the application falls below a processing rate of less than 10 transactions per second, select the optimised DLLs. Local rules apply to single local cells only and, are based on the values of object attributes and program variables at a particular time, for example, if an object attribute or local variable matches a value of 10, select the DLL requiring the value 10. The global and local rules required for a given application are particular to that application, although some rules may be common to applications, for example, memory and processor usage rules.

In addition to rules for selecting DLLs, there is also a need to provide the data that allow a local cell to physically locate and load a DLL that has been selected for execution. The locate-and-load data are packaged as part of the rules for each DLL.

The locate-and-load data, as well as the rules, are held in an Access database file, and there is one file for each local cell. When a local cell is initially embedded in an object, the database of rules contains only the locate-and-load data for each DLL. The rules do not need to have conditions for use since there is only one DLL that can possibly be executed and selection does not have to be made. Throughout the lifetime of the application, as needs and goals change, rules can be dynamically added

and modified to accommodate conditions, and more entries can be added for new DLLs.

E. ADAF Local Cell Monitoring and Reasoning

The library and rules form only two of the elements in local cells. The local monitoring and local reasoning elements are also part of the local cell. In the field of adaptive software, these functions are collectively known as reflection. Reflection is two-fold, comprising introspection and intercession. Introspection involves observing behaviour and is equivalent to monitoring. Intercession involves acting on observations by modifying behaviour and is equivalent to reasoning [24].

The local monitoring element observes the behaviour of an application by collecting runtime data, to ensure it is running correctly and is meeting goals and needs. The runtime data collected describe the actual behaviour of the application. It is then the work of the reasoning element to evaluate that data, using it as the basis for deciding which DLLs (from the library) to select for execution.

Since the rules stored in the database describe the conditions under which DLLs should execute, the reasoning element must compare actual conditions (that is, monitored data) against required or desired conditions (that is, rules in the database) and find the DLL that most closely matches the needs of actual conditions. On finding a match, the reasoning element then retrieves the locate-and-load data for the selected DLL from the database and uses those to execute the DLL.

The local monitoring and reasoning elements of the local cell are implemented in a single DLL, the local cell DLL. A DLL is chosen to implement these elements for the same reasons that DLLs are chosen for the object code. The monitoring and reasoning elements are not expected to remain static, they are expected to change and evolve over time, to meet the changing needs of the application. Therefore, like the object code, they too must be implemented in a way that will allow dynamic linking.

When a local cell is initially embedded in an object, the monitoring element may or may not need to collect runtime data, and the reasoning element may or may not need to perform any evaluation. However, throughout the lifetime of an application, as needs and goals change, additional and alternative monitoring and reasoning code can be dynamically added and modified. The specific reasoning and monitoring techniques and approaches contained within the reasoning and monitoring elements can take many forms and will vary from application to application, depending on the aspects of the application deemed important and necessitating observation, and on the autonomic activities that may be required by the application such as self-repair or self-optimisation.

F. ADAF Global Cell

Although the ADAF operates largely at a local level, with local cells embedded into application objects, there is also a global aspect to the ADAF. This global aspect is realised through the global cell, of which only one exists for any given application. The global cell contains two major elements: the evolution element and the monitoring and reasoning element. The monitoring and reasoning element collects and reasons about runtime data that cannot be collected or reasoned about at the local (module) level.

The evolution element allows dynamic adaptation and change of the application over time. The global cell must facilitate evolution of code in a way that will not disrupt system execution. Without this capability, the cells of the ADAF which compound to achieve autonomic computing, have little value. While dynamic linking plays an important role in achieving evolution, it alone is not sufficient. Indeed on its own, the use of dynamic linking is not novel or innovative. What is innovative, however, is the way in which it is exploited in the ADAF as a means of enabling autonomic software: the way it is used to structure the cells of the ADAF; the way it is used as the foundation upon which the cells of the ADAF rest; and the way it is used to achieve software evolution.

The evolution element is created through the interrelatedness between: a) the ADAF cell infrastructure; b) and the evolutionary DLL, which is the implementation of the evolution element in the global cell. The cell infrastructure of the ADAF combines the use of method DLLs with a form of proxy code, to separate adaptation code from implementation code, and to redirect code. As discussed, the implementation code of an object is separated into method DLLs in the object's library. The object itself becomes a shell that acts as a proxy, redirecting the method calls made to it, towards the local cell DLL. The local cell DLL in turn contains the elements necessary for monitoring, reasoning and selection of method DLLs for execution. This infrastructure lays the foundation for evolution, while the actual mechanism of evolution is carried out by the evolutionary DLL.

The evolutionary DLL is implemented as an event-driven thread. The thread sleeps until it receives the signal to wake up and execute its code. Once awake, the thread searches a specified temporary storage area and copies of all the DLLs it finds there into the appropriate object code libraries. The copied DLLs might contain code completely new to the application or they might be replacements for existing DLLs. The temporary storage area is populated by code developers on-the-fly, while the application is executing, with any new or updated code that is necessary. Once copying is complete, the thread deletes everything from the temporary storage area and goes back to sleep. All of this code is executed dynamically, while the application is executing and as such, new and changed code is dynamically integrated into the running application.

In addition to local monitoring and reasoning, it is necessary for monitoring and reasoning to take place at a global level, to observe runtime behaviour that cannot be observed at a local level, and to make decisions that cannot be made at a local level. The notion of global monitoring and reasoning might appear to contradict the features of self-organisation in which all such activity is localised. However, it is worth noting that a software application (even with the ADAF embedded) can never

strictly be a self-organising system and that there are some behaviours and actions that must be recognised at an application-wide (global) level to be meaningful. In other words, the application has specific objectives, not merely emergent characteristics.

Like the evolution element, the global monitoring and reasoning element of the global cell is implemented in a single DLL, the Monitor DLL, which contains its own thread of execution. The thread remains in a sleep state and at specified time intervals it is signalled to wake up and execute. The specifics of what code it executes when it does wake up, such as the data it collects, the reasoning by which the data are evaluated, the actions taken based on the reasoning, and the mechanisms by which these activities are implemented and to what extent, will be entirely dependent on the aspects of the application deemed important and necessitating observation and on the autonomic activities that may be required by the application such as self-repair or self-optimisation. In fact, similar to local cells when the ADAF is initially embedded in an application, the global monitoring and reasoning DLL may not necessarily contain any functionality. Rather, it is simply put in place to provide a facility for dynamically adding and modifying whatever future global monitoring and reasoning code might be needed, whenever it might be needed.

IV. Case Study

With the ADAF developed, it was necessary to conduct testing to determine if the ADAF enabled autonomicity in software applications. The Ship Radar Translator (SRT) application was selected as the test application because it is a self-contained, non-distributed application, with object-level granularity. The SRT is a simplified version of a real-world application. It receives encoded messages, in the form of character strings, from the radar on a ship and translates these messages into a meaningful form which is then displayed on a visual display unit and written to a database file.

The SRT is made up of ten C++ classes and has approximately 7,500 lines of code. When the SRT was developed using the ADAF, which is an autonomic-oriented development process, the result was the ADAF-SRT application. The ADAF-SRT became a test application and functioned in exactly the same way as the SRT application.

Test scenarios were developed for the ADAF-SRT to demonstrate autonomic capabilities. One of the test scenarios demonstrated self-configuration and self-evolution where monitoring and reasoning were local [35]. A second test scenario was developed to demonstrate self-diagnosis and self-monitoring, in which memory usage was monitored and appropriate action initiated when a memory leak was detected. In this scenario, monitoring and reasoning were global. Space precludes discussion of both scenarios, so only the second is detailed here.

A. Test Scenario Demonstrating Self-diagnosis and Self-monitoring

It is reasonable to expect that autonomic software should be able to dynamically detect problems or errors that may occur in an application and to take preventative or corrective action. The occurrence of problems and errors in software, resulting in software failure, is prevalent in deployed applications. For this reason, self-healing or self-repair is a strong focus of autonomic computing research. Since the research in this paper does not focus on software failure or self-repair per se, but rather, focuses on a more holistic autonomic computing solution, it was sufficient for this test scenario to demonstrate dynamic error detection, without extending that as far as healing or repairing from the error.

As such, the test scenario demonstrates the dynamic detection of a specific type of error, namely a memory leak that went undetected at development time. In the scenario, the ADAF-SRT application's memory usage was monitored to collect data, and then the data used to reason about whether there was a memory leak. Action was taken in the event where the leak became severe enough to potentially affect the performance of the application.

In preparation for conducting the test scenario, it was necessary to carry out a substantial amount of preparatory work: the memory leak was created; instrumentation code for monitoring was decided upon; reasoning code was decided upon; and the monitoring and reasoning code was integrated into the application by dynamically changing Monitor DLL (that is, the ADAF DLL responsible for global monitoring and reasoning).

B. Preparing for the Test Scenario: Creating the Memory Leak

While many aspects of an application could be monitored, it was decided to monitor the application for memory leaks. A memory leak occurs in an application when memory is allocated but not freed up and returned to the operating system when no longer in use. Memory leaks are often difficult to detect during development and might not be noticed until deployment, at which time they can be detrimental to an application.

There are several causes of memory leaks and, for this test scenario it was decided to inject memory leaks by including the *new* operator without using its corresponding *delete* operator. For the purposes of testing, a simple dummy class, called CMemoryLeak, was written, and a .NET Windows Forms application called LeakyApp was created with deliberately inserted memory leaks, whereby pointers to CMemoryLeak objects were created using the *new* operator without the corresponding *delete* operator.

C. Preparing for the Test Scenario: Developing Monitoring Code

The instrumentation method selected to monitor data was sampling, since it is less intrusive and carries much less overhead than the alternative probe methods. The Performance Monitor, or PerfMon, was used to gather the data samples. PerfMon is a performance profiler built-in

to the Windows NT operating system. It allows a range of behaviours to be monitored, capturing samples of the raw data being generated by components and objects in the host machine and running applications. The raw data samples collected by PerfMon are captured into objects called performance counters which are expressed as numbers.

PerfMon has over a thousand predefined performance counters that monitor a host of machine and application behaviours. These are grouped into fifty-four performance counter categories, where each category refers to a specific area of machine functionality such as the processor, threading, memory, and disk operation. In addition, there are a number of .NET classes which allow performance counters to be manipulated and accessed programmatically.

A dummy .NET application was created and used along with the LeakyApp application to produce performance counter values that could be observed and interpreted, thus provide a starting point for isolating those performance counters relevant for monitoring memory leakage. After considerable time and effort, counters relevant to memory leakage were eventually isolated to Memory\AvailableMBytes and Process\PrivateBytes.

Process\PrivateBytes provides data about individual processes on a machine, for example, an executing application. It is the current size of RAM, in bytes, that a process has been allocated and that cannot be shared with other processes. The threshold for PrivateBytes depends on the application and on the settings in the host machine's configuration, but it should not exceed AvailableMBytes when the machine is idle.

Memory\AvailableMBytes provides data about the host machine (not individual processes running on the machine). It is the total amount of physical RAM, in megabytes (MB), available to processes running on the host machine. According to Microsoft Developer Network (MSDN) recommendations, AvailableMBytes should be at least 10% of the total RAM of the host machine, and ideally should exceed 20% [36].

If AvailableMBytes is observed to maintain a consistent value of less than 20-25% of RAM, it indicates that the host machine is low on RAM, caused either by memory limitations or by an application that is not releasing memory. To determine if the problem is due to memory not being released, that is a memory leak, it is further necessary to observe PrivateBytes for each process running on the machine. If this counter is not rising for any application, then the problem is due to lack of RAM. However, if it is noted that for a particular application, this counter is increasing without ever going constant, then it is a definite indicator of a memory leak, since the longer a memory leaking application runs, the more memory it uses and thus the more PrivateBytes it needs.

D. Preparing for the Test Scenario: Developing the Reasoning Code

The technique selected for reasoning was fuzzy logic. Fuzzy logic is routinely used in situations where there is a continuum of fuzzy, non-sharp values rather than crisp, precise values, for example, control systems in manufacturing and production environments. The memory leak scenario is similar to a control system because a memory leak exists, to a greater or lesser degree, along a continuum from no memory leak to a severe memory leak. Therefore, fuzzy logic is ideal for this scenario, and was applied by developing a fuzzy inference system in MATLAB. Developing the fuzzy inference system required the completion of two steps: acquiring archetypal data; modelling the relevant data.

To acquire the data, the behaviour of PrivateBytes and AvailableMBytes was observed by collecting their data values through a series of executions of the ADAF-SRT application, and the data values were used to identify patterns in behaviour. On completion of the series of executions, with ever increasing levels of memory leak introduced at each iteration, there was a total of 1,110 data values. These data were imported into MATLAB for analysis where they were plotted onto graphs. The graphs provided a means of observing trends and patterns in the application's memory usage for times when there was no memory leak, for when there were low level leaks, for when there were moderate leaks, and for when there were severe leaks.

From the plotted data, it was observable that when the application was executing with no memory leak, both PrivateBytes and AvailableMBytes remained at relatively constant and static levels, although PrivateBytes was seen to decrease slightly at times, while AvailableMBytes sometimes increased. The slight increases or decreases observed were tiny fluctuations only, not continuous increases or decreases, and they were brought about by varying intensities of processing activity in the application. When a low level memory leak was introduced to the application, PrivateBytes showed a slight, but steady increase, and AvailableMBytes showed a slight, but steady decrease. As greater levels of memory leak were introduced, PrivateBytes increased ever further while AvailableMBytes progressively decreased.

With the PrivateBytes and AvailableMBytes data acquired and analysed, it was next necessary to model the data by building a fuzzy inference system. The system had two inputs, PrivateBytes and AvailableMBytes, and one output, MemoryLeak, since it is PrivateBytes and AvailableMBytes together that are needed to determine if there is a memory leak. The system also included a set of rules to evaluate the inputs to produce the output value. Using the plotted data collected above, the PrivateBytes range on the fuzzy inference system was divided into three levels: low, medium and high, where *low* represented no memory leak in the application; *medium* represented a moderate memory leak; and *high* represented a severe memory leak.

Using the plotted data and MSDN recommendations (that AvailableMBytes should not fall below 10% of RAM and ideally should stay above 20% of RAM), the range for AvailableMBytes was divided into three levels: low, medium, and high, where *low* indicated that there was a memory shortage on the host machine; *medium*

indicated that there was an adequate amount of memory; and *high* indicated that there was a healthy amount of memory. The output for the fuzzy inference system, MemoryLeak, was modelled. The range for MemoryLeak, that is, the level of memory leakage, was represented on a scale that went from 0-12, and was divided into three levels: *no memory leak*, *moderate* memory leak, and *severe* memory leak.

To complete the fuzzy inference system, a set of nine rules were created, and these are shown in Fig. 3. The rules defined all combinations of the three levels, low, medium and high, in PrivateBytes and AvailableMBytes with their resulting MemoryLeak. When PrivateBytes is low and AvailableMBytes is high, there is enough available RAM on the host machine and the application is using an acceptable amount of its allocated memory. Therefore, there is no memory leak. When PrivateBytes is medium and AvailableMBytes is high, there is enough available RAM on the host machine and although the application is using more of its allocated memory, the problem is still not considered a memory leak. When PrivateBytes is high and AvailableMBytes is high, there is enough available RAM on the host machine but the application is using more of its allocated memory than is acceptable, and a moderate memory leak is flagged.

When PrivateBytes is low and AvailableMBytes is medium, there is less, but still adequate, memory available on the host machine and the application is using an acceptable amount of its allocated memory. Therefore, there is no memory leak. When PrivateBytes is medium and AvailableMBytes is medium, the host machine is running on less though adequate memory, while the application is using more of its allocated memory, resulting in a moderate memory leak. When PrivateBytes is high and AvailableMBytes is medium, there is adequate machine memory available but the application is using more of its allocated memory than is acceptable, and a moderate memory leak is flagged.

When PrivateBytes is low and AvailableMBytes is low, the memory on the host machine has fallen below a healthy level but the application is using acceptable amount of its allocated memory. Therefore, there is no memory leak. When PrivateBytes is medium and AvailableMBytes is low, machine memory is below an acceptable level and the application is using more of its allocated memory, and a moderate memory leak is flagged.

When PrivateBytes is high and AvailableMBytes is low, machine memory is below an acceptable level and the application is using more of its allocated memory than is acceptable. This situation may quickly lead to serious problems such as highly degraded application performance or a system crash, so the problem is flagged as a severe memory leak.

E. Preparing for the Test Scenario: Dynamically Integrating the Monitoring and Reasoning Code

With the monitoring and reasoning code determined, it was then necessary to integrate that code into the ADAF-SRT application. This involved making changes to a part of the global cell of the ADAF, namely the Monitor DLL.

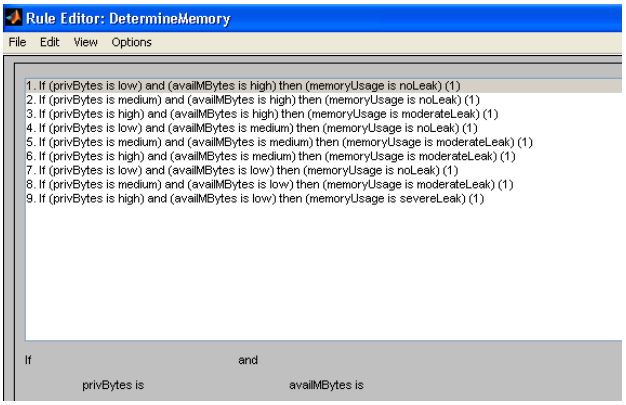

Figure 3. Rules in the fuzzy inference system.

The instrumentation code necessary for monitoring memory usage at intervals was added to Monitor DLL. The code necessary to execute the fuzzy inference system at intervals was added to Monitor DLL.

Further code was added to Monitor DLL to allow appropriate action to be taken, depending on the result from the fuzzy system. If the fuzzy system showed that the memory leak value was between 0 and 3, there was no memory leak and no action would be taken. If the value from the fuzzy system was between 4 and 7, there was a moderate memory leak and a warning would be written to a log called Problems.log. This warning took the form of a message indicating the time at which the warning was detected, and providing a list of DLLs that might potentially be the source of the memory leak.

If the value from the fuzzy system was between 8 and 10, there was a severe memory leak, and an error was written to the Problems.log. The error took the form of a message indicating the time at which the error was detected and providing a list of DLLs that might potentially be the source of the memory leak. Further, in the case where a previous version of the application existed, a rollback to that previous version would be performed. The rollback was based on the assumption that a previous version of the application would not have a memory leak and that the most recent version must be the cause of the memory leak. Therefore, rolling back to a previous version would ensure that the most recent, memory-leaking version could be avoided.

V. Case Study Results

On completion of the preparatory work for the test scenario, the scenario itself was conducted and the results observed. This took place in three distinct parts whereby the ADAF-SRT application was executed with a) no memory leak; b) a moderate memory leak; and c) a severe memory leak.

A. Conducting the Test Scenario: Results for Part a)

For part a) of the test scenario, the values for PrivateBytes remained *low* for the first 6 hours of execution, and the values for AvailableMBytes remained *high*. After 6 hours of execution, when LeakyApp was executed multiple times, the values for AvailableMBytes

began to decrease to *medium*, while the values for PrivateBytes remained *low*. Continued execution of LeakyApp caused AvailableMBytes to decrease to *medium* at 6 hours 45 minutes, and then drop to *low* at 8 hours 30 minutes.

In summary, PrivateBytes remained *low* all the way throughout execution, and AvailableMBytes moved from *high*, then to *medium*, and finally to *low*. When the application was shut-down, the Problems.log was checked and found to be empty, indicating that a memory leak had not been detected. A MATLAB graph plotting all of the data values collected for AvailableMBytes during part a) of the test scenario is provided in Fig. 4, and Fig. 5 shows a MATLAB graph plotting all of the data values collected for PrivateBytes.

In addition to the data values collected while conducting part a) of the test scenario, other data were collected: the time taken to collect the performance counter data values at each interval was measured and found to be negligible; the average time taken to open, execute, and close MATLAB at each interval was measured and found to be 1.02 seconds; the time taken to check for a memory leak at each interval was measured and found to be negligible.

B. Conducting the Test Scenario: Results for Part b)

In part b) of the test scenario, the values for PrivateBytes remained *low* for the first 6 hours of execution, although the memory leak was causing the values to increase slightly. The values for AvailableMBytes were *high*, and although the memory leak did cause the values to decrease slightly, the reduction was not enough to lower the values from *high*.

After this point, when LeakyApp was executed multiple times, the values for AvailableMBytes began to drop steadily and the values for PrivateBytes continued to rise. At 6 hours 20 minutes, PrivateBytes moved into *medium*, and at 6 hours 25 minutes the values for AvailableMBytes moved into *medium*. At this point, a moderate memory leak was detected and a warning message was written to the Problems.log. Continued execution of LeakyApp caused AvailableMBytes to decrease further until it reached *low*, at 8 hours 40 minutes. PrivateBytes remained at *medium*. Throughout this, the moderate memory leak continued to be detected and warning messages written to the Problems.log.

To summarise, PrivateBytes started at *low* and moved to *medium* where it stayed for the remainder of execution. AvailableMBytes moved from *high*, then to *medium*, and finally to *low*. A MATLAB graph plotting all of the data values collected for AvailableMBytes during part b) of the test scenario is provided in Fig. 4, and Fig. 5 shows a MATLAB graph plotting all of the data values collected for PrivateBytes.

In addition to the data values collected while conducting part b) of the test scenario, other data were collected: the time taken to collect the performance counter data values at each interval was measured and found to be negligible; the average time taken to open, execute, and close MATLAB at each interval was measured and found to be 1.02 seconds; the time taken to check for a memory leak at each interval was measured and found to be negligible; the time taken to write a warning message to the Problems.log was measured and found to be negligible.

C. Conducting the Test Scenario: Results for Part c)

For part c) of the test scenario, the values for PrivateBytes remained *low* – and rising – for the first 5 hours of execution, when at that point, the values moved to *medium*. AvailableMBytes values started at *high*, although the memory leak caused the values to decrease steadily. At 5 hours 15 minutes, AvailableMBytes values moved to *medium*. At this point, a moderate memory leak was detected and the first warning message was written to the Problems.log. After 6 hours 25 minutes, PrivateBytes moved to *high*, while AvailableMBytes remained *medium*.

LeakyApp was executed multiple times, causing the AvailableMBytes values to drop. PrivateBytes remained *high*. LeakyApp continued to be executed and at 7 hours 40 minutes the values for AvailableMBytes dropped to *low*, while PrivateBytes remained *high*. Throughout this, the moderate memory leak continued to be detected and warning messages written to the Problems.log. A MATLAB graph plotting all of the data values collected for AvailableMBytes during part c) of the test scenario is provided in Fig. 4, and Fig. 5 shows a MATLAB graph plotting all of the data values collected for PrivateBytes.

At this point, that is, 7 hours 40 minutes, with PrivateBytes at *high* and AvailableMBytes at *low*, a severe memory leak was detected. An error message was written to the Problems.log, and an application rollback was performed where the application was returned to a previous version which did not contain the memory leak. Specifically, the rollback meant a return to all the previous versions of the local cell DLLs, and therefore, the DLL with the injected memory leak stopped being executed, and an earlier, non-leaking version of it began being executed instead.

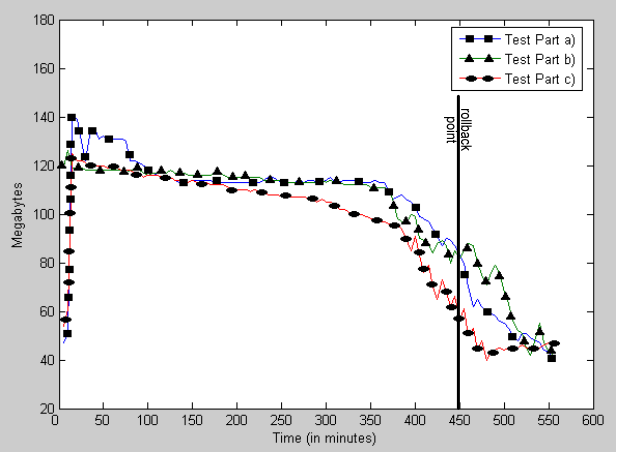

Figure 4. Plot of data collected for AvailableMBytes during parts a), b) and c) of the test scenario

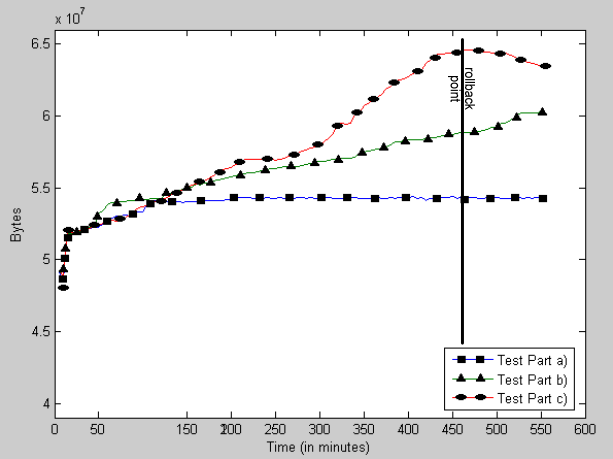

Figure 5. Plot of data collected for PrivateBytes during parts a), b) and c) of the test scenario

The time of the rollback of the application is visible from Figs. 4 and 5, where it is marked on both figures with a vertical line. Notice that towards the end of execution for part c) data, PrivateBytes values were visibly falling, while AvailableMBytes values were on the increase, indicating that the rollback was beginning to take effect and that the application had stopped leaking memory. The rollback also prevented any further warning or error messages to be written to the Problems.log.

When the application was shut-down, the Problems.log was checked and found to have 11 warning messages logged, and 1 error message logged. A snapshot of the warning and error messages is shown in Fig. 6. It is worth noting that in the snapshot, one of the DLLs identified as having a potential memory leak is HandleRadarMsg, and this is in fact the memory-leaking DLL. Since the instrumentation technique used was sampling, it was more difficult to pinpoint the precise DLL causing the memory leak, and a list of the potential problem DLLs was all that was provided.

This may be acceptable, depending on the needs of a given application. However in the situation where greater precision is required, the instrumentation technique could be changed to use probes, whereby every DLL containing memory allocation through the *new* operator could be instrumented. In this way, the exact DLL (or DLLs) that is the source of the memory leak could be pinpointed. Given the design of the ADAF, such changes could be easily accommodated by dynamically replacing original, non-instrumented versions of the DLLs with new, instrumented versions.

In addition to the data values collected while conducting part c) of the test scenario, other data were collected: the time taken to collect the performance counter data values at each interval was measured and found to be negligible; the average time taken to open, execute, and close MATLAB was measured and found to be 1.02 seconds; the time taken to check for a memory leak and write a warning or error message to the Problems.log was measured and found to be negligible; the time taken to perform the application rollback was measured and found to be 0.47 seconds.

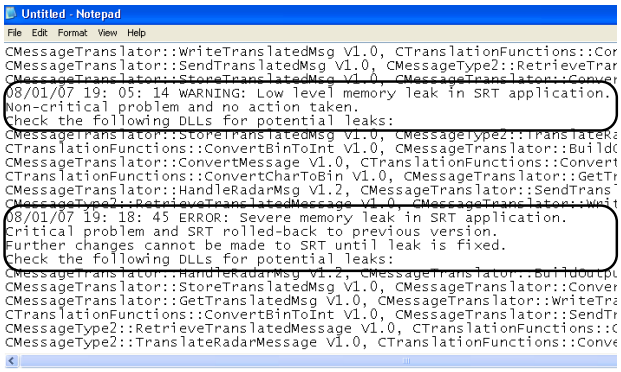

Figure 6. Snapshot of warning and error messages logged in Problems.log for part c) of the test scenario

VI. SUMMARY

The ADAF was presented as an autonomic-oriented development process for achieving autonomicity in software applications whereby high-level self-organising features were identified and then integrated into the framework Developed for use with object-oriented software applications, ADAF was embedded into a sample real-world application, the Ship Radar Translator, and was tested against a scenario that would determine if the application could exhibit the autonomic capabilities of self-monitoring and self-diagnosis.

Test results demonstrated that the ADAF-SRT application was able to achieve significant levels of autonomicity in terms of self-monitoring and self-diagnosis. The original SRT application has no such capability. While it would not be impossible to add that kind of capability to the SRT, it would have to be done by halting execution and then rebuilding. Additionally, if a memory leak were found and had to be corrected, that work would have to be conducted while the application was stopped, for there is no way to separate problem areas of the application and dynamically correct them.

ADAF adds a level of complexity that does not exist in the original application. However the complexity brings the advantages of autonomic capabilities and when managed in an efficient and controlled manner, means the application is no different from any other complex application. Future work is directed at issues of scalability and performance, and the implementation of a more advanced reasoning system.

REFERENCES

[1] E. P. Kasten and P. K. McKinley, "MESO: Supporting Online Decision Making in Autonomic Computing Systems", *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 4, 2007, pp. 485-499.

[2] R. Gerber, A. J. C. Bik, K. B. Smith and X. Tian, *The Software Optimisation Cookbook*, Intel Press, second edition, 2006.

[3] W. Wayt-Gibbs, "Autonomic Computing", [Online], *Scientific American*, 6th May, 2002. Available at: http://crash.stanford.edu/articles/sciam1/autonomic.html/[accessed November 2003].

[4] P. Horn, "Why Autonomic Computing will Reshape IT", [Online], *ZDNet Magazine*, 16th October, 2001. Available

at: http://www.zdnet.com.au/newstech/enterprise/story/0,2000048640,20261187-1,00.htm/[accessed November 2003].

[5] M. Mamei, R. Menezes, R. Tolksdorf and F. Zambonelli, "Case Studies for Self-organization in Computer Science", *Journal of Systems Architecture*, vol. 52, iss. 8-9, 2006, pp. 443-460.

[6] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing", *IEEE Computer*, January, 2003, pp. 41-50.

[7] A. G. Ganek and T. A. Corbi, "The Dawning of the Autonomic Computing Era", *IBM Systems Journal*, vol. 42, no. 1, 2002, pp. 5-18.

[8] T. Hall, "Autonomic Computing: It's about making Smarter Systems", [Online], *LDD Today*, 2nd June, 2003. Available at: http://www-10.lotus.com/ldd/today.nsf/9148b29c86ffdcd385256658007aaa0f/337bfa71918408a685256d330047cbbe?OpenDocument/[accessed February 2004].

[9] D. Patterson et al, "Recovery Oriented Computing (ROC): Motivation, Definition, Techniques and Case Studies", [Online], *UC Berkeley Computer Science Technical Report* UCB//CSD-02-1175, 2002. Available at: http://roc.cs.berkeley.edu/papers/ROC_ TR02-1175.pdf/ [accessed November 2003].

[10] G. Candea and A. Fox, "Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel", [Online], *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, 2001. Available at: http://roc.cs.berkeley.edu/papers/ recursive_restartability.pdf/[accessed November 2003].

[11] S. Brueckner, G. di Marzo Serugendo, D. Hales and F. Zambonelli, "*Engineering Self-organising Systems, 3rd International Workshop"*, 2005, Springer: Berlin, Heidelberg.

[12] M. Sipper, "An Introduction to Artificial Life", *AI Expert Special Issue: Explorations in Artificial Life*, September 1995, pp. 4-8.

[13] M. Kim, J. Jeong and S. Park, "From Product Lines to Self-managed Systems: An Architecture-based Runtime Reconfiguration Framework", *SIGSOFT Software Engineering Notes*, vol. 30, no. 4, 2005, pp. 1-7.

[14] M. A. Munawar and P. A. Ward, "Better Performance or Better Manageability?", *Proceedings of the 2005 Workshop on Design and Evolution of Autonomic Application Software*, St. Louis, USA, 21st May, 2005, pp. 1-4.

[15] C. Boulton, "IBM Spruces up Autonomic Computing Offerings", [Online], *ASPNews*, 7th March, 2003. Available at: http://www.aspnews.com/news/article.php/2106371/ [accessed November 2003].

[16] R. Sterritt and D. Bustard, "Towards an Autonomic Computing Environment", [Online], *14th International Workshop on Database and Expert Systems Applications*, Prague, September, 2003. Available at: http://www.infj.ulst.ac.uk/~roy/papers/autonomic/ieee_dexa_2003_ace.pdf/[accessed November 2003].

[17] M. Mesnier, E. Thereska, G. R. Ganger, D. Ellard and M. Seltzer, "File Classification in Self-*storage Systems", *Proceedings of the International Conference on Autonomic Computing*, New York, USA, 17th-18th May, 2004, pp. 44-51.

[18] J. Martin-Flatin, J. Sventek and K. Geihs, "Self-Managed Systems and Services: Introduction", *Communications of the ACM*, vol. 49, no. 3, 2006, pp 37-39.

[19] M. Jelasity, O. Babaoglu and R. Laddaga, "Self-management through Self-organisation", *IEEE Intelligent Systems*, March/April, 2006, pp. 8-9.

[20] F. Heylighen and C. Gershenson, "The Meaning of Self-organisation in Computing", *IEEE Intelligent Systems*, vol. 18, no. 4, July/August, 2003, pp. 27-86.

[21] G. di Marzo Serugendo et al, "Self-organisation: Paradigms and Applications", [Online], *Engineering Self-Organising Applications Working Group*, 2004. Available at: http://www.agentcities.org/Activities/WG/ESOA/[accessed January 2005].

[22] B. Meehan, G. Prasad, T. M. McGinnity, "A Framework for Autonomic Software", *Proceedings of IEEE International Conference on Systems*, Man, *and Cybernetics*, October, 2007, Montreal, Quebec, Canada.

[23] S. Cheng et al, "Using Architectural Style as a Basis for System Self-repair", [Online], *Software Architecture: System Design, Development, and Maintenance*, Kluwer Academic Publishers, 2002. Available at: http://www2.cs.cmu.edu/afs/cs.cmu.edu/project/cmcl/archive/Libra-papers/WICSA02.pdf/[accessed November 2003].

[24] P. K. McKinley, S. M. Sadjadi, E. P. Kasten and B. H. C. Cheng, "Composing Adaptive Software", *IEEE Computer*, vol. 37, iss. 7, 2004, pp. 56-64.

[25] J. Keeney and V. Cahill, "Chisel: A Policy-driven, Context-aware, Dynamic Adaptation Framework", *Proceedings of the 4th IEEE international Workshop on Policies for Distributed Systems and Networks*, Washington DC, USA, 4th-6th June, 2003, pp. 3-14.

[26] D. Ardagna, M. Comuzzi, E. Mussi, B. Pernici and P. Plebani, "PAWS: A Framework for Executing Adaptive Web-service Processes", *IEEE Software*, vol. 24, no. 6, 2007, pp. 39-46.

[27] S. D. Fleming, B. H. Cheng, R. E. Stirewalt and P K. McKinley, "An Approach to Implementing Dynamic Adaptation in C++", *SIGSOFT Software Engineering Notes*, vol. 30, no. 4, 2005, pp. 1-7.

[28] K. H. Fung, G. Low and P. K. Bay, "Embracing Dynamic Evolution in Distributed Systems", *IEEE Software*, vol. 21, iss. 2, 2004, pp. 49-55.

[29] M. Hicks, J. Moore and S. Nettles, "Dynamic Software Updating", [Online], *University of Maryland, Programming Languages Research*, 2001. Available at: http://citeseer.ist.psu.edu/cache/papers/cs/22700/http:zSzzSzwww.cis.upenn.eduzSz~mwhzSzpaperszSzdyn_update.pdf/hicks01dynamic.pdf/[accessed November 2003].

[30] G. Hamilton and S. Radia, "Using Interface Inheritance to Address Problems in System Software Evolution", *Proceedings of the ACM Workshop on Interface Definition Languages*, Portland, Oregon, USA, 1994, pp. 119-128.

[31] T. C. Goldstein and A. D. Sloane, "The Object Binary Interface – C++ Objects for Evolvable Shared Class Libraries", [Online], *Technical Report TR-94-26*, Sun Microsystems Laboratories Incorporated, 1994. Available at: http://research.sun.com/techrep/1994/smli_tr9426.pdf /[accessed July 2004].

[32] G. Hjálmtysson and R. Gray, "Dynamic C++ Classes: a Lightweight Mechanism to Update Code in a Running Program", In *Proceedings of the USENIX Annual Technical Conference*. 1998.

[33] N. Jennings, K. Sycara and M. Wooldridge, "Roadmap of Agent Research and Development", *Autonomous Agents and Multi-Agent Systems*, vol. 1, no. 1, 1998, pp. 7-38.

[34] N. Jennings, "An Agent-based Approach for Building Complex Software Systems", *Communications of the ACM*, vol. 44, no. 4, 2001, pp. 35-41.

[35] B. Meehan, G. Prasad, T. M. McGinnity, "Autonomic Software through Self-organisation", *IT&T Conference 2006, Technical Session on Software Innovations*, Carlow, Ireland, October 25-26, pp. 101-109, 2006.

[36] MSDN Library, "Measuring .NET Application Performance (Chapter 15)", [Online], *MSDN .NET Development: Improving .NET Applications Performance and Scalability*, 2004. Available at: http://msdn2.microsoft.com/en-us/library/[accessed July 2005].

Bridget Meehan is a PhD student in the School of Computing and Intelligent Systems at the Magee Campus of the University of Ulster. She holds a BSc in Applied Computing from the University of Ulster earned in 1992, and an MSc in Computer Science from the University of Limerick earned in 1998.

Girijesh Prasad is a lecturer and researcher in the School of Computing and Intelligent Systems at the Magee Campus of the University of Ulster. Previously he worked as a Digital Systems Engineer in Uptron India Ltd, Lucknow, India, and then as a Power Plant Engineer in a thermal power station of UPSEB at Obra, India.

He also worked as a Research Fellow on an UK EPSRC/industry funded research project at Queen's University of Belfast. He holds a first class honours bachelor degree in Electrical Engineering and a first class honours master degree in Computer Science and Technology. He obtained his PhD degree from the Queen's University of Belfast. He is also a Chartered Engineer and a member of IEE.

T. M. McGinnity has been a member of the University of Ulster academic staff since 1992. He holds the post of Professor of Intelligent Systems Engineering within the Faculty of Computing and Engineering. He holds a first class honours degree in physics, and a doctorate from the University of Durham. He is a Fellow of the IEE, member of the IEEE, and a Chartered Engineer. He has 28 years experience in teaching and research in electronic and computer engineering, and was formerly Head of School of Computing and Intelligent Systems at the University's Magee campus. He is currently Director of the Intelligent Systems Research Centre and Director of the University's technology transfer company, UUTech. He is the author or co-author of over 175 research papers, and has been awarded both a Senior Distinguished Research Fellowship and a Distinguished Learning Support Fellowship by the University.