INVESTIGATION OF A TELEO-REACTIVE APPROACH FOR THE DEVELOPMENT OF AUTONOMIC MANAGER SYSTEMS

JAMES HAWTHORNE

A thesis submitted in partial fulfilment of the requirements of the University of Greenwich for the Degree of Doctor of Philosophy

April 10, 2013

A small proportion of the work presented here may have been taken from my own publications. Any reproduced material is entirely my own work however. Following is a list of my publications:

J. Hawthorne and R. Anthony, "Using a teleo-reactive approach in building self-managing systems," *International Journal of Autonomous and Adaptive Communication Systems*, vol. 5, no. 3, pp. 255-273, Jul. 2012.

J. Hawthorne, R. Anthony, and M. Petridis, "Improving the development process for teleo-reactive programming through advanced composition," *The Seventh International Conference on Autonomic and Autonomous Systems (ICAS 2011)*, Venice/Mestre, Italy, May 2011, pp. 75-80.

J. Hawthorne and R. Anthony, "Developing teleo-reactive autonomic solutions," *International Journal of Computer and Information Science (IJCIS)*, vol. 11, no. 3, pp. 26-34, 2010.

J. Hawthorne and R. Anthony, "A methodology for the use of the teleo-reactive programming technique in autonomic computing," *Software Engineering Artificial Intelligence Networking and Parallel/Distributed Computing (SNPD 2010)*, 11th ACIS International Conference, pp. 245-250, June 2010.

J. Hawthorne and R. Anthony, "Using a teleo-reactive programming style to develop self-healing applications," *Autonomics 2009: Third International ICST Conference Autonomic Computing and Communication Systems*, Limassol, Cyprus, September 2009.

P. Ward, M. Pelc, J. Hawthorne, and R. Anthony, "Embedding dynamic behaviour into a self-configuring software system," *ATC 08: Proceedings of the 5th international conference on Autonomic and Trusted Computing*, Berlin, Heidelberg, Springer-Verlag, 2008, pp. 373-387.

J. Hawthorne and R. Anthony, "A reconfigurable component model using reflection," *SERENE 08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems*, Newcastle, ACM, November 2008, pp. 95-100.

R. Anthony, M. Pelc, P. Ward, and J. Hawthorne, "Flexible and robust run-time configuration for self-managing systems," *Self-Adaptive and Self-Organizing Systems*, Second IEEE International Conference, 2008, pp. 491-492.

R. Anthony, M. Pelc, P. Ward, J. Hawthorne, and K. Pulnah, "A run-time configurable software architecture for self-managing systems," *International Conference on Autonomic Computing*, Los Alamitos, CA, USA, IEEE Computer Society, June 2008, pp. 207-208.

R. Anthony, P. Ward, D. Chen, A. Rettberg, J. Hawthorne, M. Pelc, and M. Törngren, "A middleware approach to dynamically configurable automotive embedded systems," *ISVCS*, 2008.

DECLARATION

"I certify that this work has not been accepted in substance for any degree, and is not concurrently being submitted for any degree other than that of Doctor of Philosophy being studied at the University of Greenwich. I also declare that this work is the result of my own investigations except where otherwise identified by references and that I have not plagiarised anothers work"

James Hawthorne
(Student)

Dr. Richard J. Anthony
(1st Supervisor)

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor Richard Anthony who, despite the level of research work and lecturing duties he has, and despite my constant harassment on Skype, always has time to provide me with very useful advice and feedback. I am sure that my work would be half its quality without his expert input. I also have to thank him for having faith in my abilities when he originally employed me to work on the DySCAS European project in 2006, without which I would not have progressed to this point.

My colleagues and now close friends who I worked alongside during my time as a researcher on DySCAS have been a constant source of inspiration. Mariusz who knows so much about so many things. And Paul who seems to be able to decipher even my most obscure ramblings.

Friends in the office who have kept me sane and provided me with a lot of help and inspiration including Stelios, Kabir, Thaddeus, Aihua, Cain, Muesser, Esther, Aleks and Yue.

Special thanks must also go to Peter Smith who first encouraged me to join as a PhD student, as a researcher and in various other activities. He has helped me more than he knows.

Finally, my parents deserve my thanks as they have had to put up with me even during my more difficult days. I do appreciate your support.

ABSTRACT

As the demand for more capable and more feature-rich software increases, the complexity in design, implementation and maintenance also increases exponentially. This becomes a problem when the complexity prevents developers from writing, improving, fixing or otherwise maintaining software to meet specified demands whilst still reaching an acceptable level of robustness.

When complexity becomes too great, the software becomes impossible to effectively be managed by even large teams of people. One way to address the problem is an Autonomic approach to software development. Autonomic software aims to tackle complexity by allowing the software to manage itself, thus reducing the need for human intervention and allowing it to reach a maintainable state.

Many techniques have been investigated for development of autonomic systems including policy-based designs, utility-functions and advanced architectures. A unique approach to the problem is the teleo-reactive programming paradigm. This paradigm offers a robust and simple structure on which to develop systems. It allows the developer the freedom to express their intentions in a logical manner whilst the increased robustness reduces the maintenance cost.

Teleo-Reactive programming is an established solution to low-level agent based problems such as robot navigation and obstacle avoidance, but this technique shows behaviour which is consistent with higher-level autonomic solutions. This project therefore investigates the extent of the applicability of teleo-reactive programming as an autonomic solution. Can the technique be adapted to allow a more ideal 'fitness for purpose' for autonomics whilst causing minimal changes to the tried and tested original structure and meaning? Does the technique introduce any additional problems and can these be addressed with improvements to the teleo-reactive framework?

Teleo-Reactive programming is an interesting approach to autonomic computing because in a Teleo-Reactive program, its state is not predetermined at any moment in time and is based on a priority system where rules execute based on the current environmental context (i.e. not in any strict procedural way) whilst still aiming at the intended goal. This method has been shown to be very robust and exhibits some of the qualities of autonomic software.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Autonomic software aims to reduce the need for human intervention in the development and maintenance process, shifting many tasks to the software itself where the human developer is either not needed or their input can be reduced. Autonomic software should be able to dynamically adapt to the inevitable changes in conditions; autonomic software should be able to configure, heal, optimise and protect itself.

The purpose of this automating of tasks is to reduce the complexity of software which has increased to an unmanageable level as software size has increased. In autonomics, many of the mundane tasks which are often handled by the developer are pushed into background activities leaving the developer free to focus on their goal. Any unplanned configuration task or run-time code errors for example should also be handled in the background by the autonomic services. This approach is similar to the way the autonomic nervous system of a biological system works. For example, breathing and fighting disease etc. are processed without conscious thought so other activities such as gathering food can gain more focus.

There have been many different approaches to the development of autonomic software such as policy-driven approaches, utility-function and artificial intelligence technique based ideas as well as various architecture designs incorporating dynamic adaptation techniques. The approach in this project however advocates a paradigm shift from traditional programming techniques to a teleo-reactive based approach.

This technique is used in robotics to reliably and robustly guide a robot towards the completion of a goal; but we find that many of the benefits afforded in robotics are also evident at a higher and autonomic level. The concept of using an alternative paradigm, is a unique solution to the autonomic challenge which can be considered to have many natural benefits.

The main aim is to explore the extent of teleo-reactive benefits in high level programming and autonomic fields. We argue that teleo-reactive programming can be considered as a valid autonomic technique. Whilst the teleo-reactive (T-R) technique has many benefits in both high and low-level programming, it also presents some unique challenges. This project also aims to highlight and address those challenges and much of this work is targeted at reducing these problems.

The approach taken in this project is to build T-R into a Java framework which can be used to compose applications built on a T-R methodology, examples of this are evidenced in chapter 4. The framework can then be used by any developer wishing to use T-R. The framework has two main benefits, firstly T-R programs require an interpreter which the framework provides (the framework can then be reused in many applications). Secondly, this will allow changes and adaptations to be made to the framework without any negative effects on the main development task. For example if an application uses the framework and a performance increase is made to the framework, this will have a positive impact on the application without the need for redesign on the side of the application which uses the newly improved framework. Low coupling between the framework and the application is therefore a requirement to ensure this. This is a principle that is adhered to throughout the project, any changes made to one part of software should not require additions to be made to another part in order to utilise the benefits. Nor should a change to the framework have a positive effect on one application but a related negative effect on another.

We have built several demonstrators which highlight these benefits and challenges of T-R programming. The main demonstrator shows how a T-R manager can be used to control the processes in a datacentre. There are three differently configured managers which can be alternated between, controlling the same datacentre with the same simulated data in each case. The Temporal Analysis Panel (described in section 3.2.4) together with the logged data, alert the developers early to structural and composition problems. The logged data can then be graphed and compared. This could allow the developer to compare one T-R manager against another or for early detection of defects in the running T-R manager.

1.1 Definitions

In chapter 2 we detail and critically evaluate the current state of the art. The rest of this section however, gives the reader a brief introduction to the fields and topics we will discuss.

1.1.1 Complexity

As the demand for more capable and more feature-rich software increases, the complexity in design, implementation and maintenance also increases exponentially. Complexity is not simple to define however, as there are many facets to software complexity. It is not simply the number of lines of code because if, for example, two pieces of code are fifty lines each and the second piece of code contains many more iteration and selection statements then it would likely be more complex than the first. If a piece of code contains large amounts of coupling between objects so that changes in one part of code requires changes in many other places, that too adds to complexity. Also, if sections of code are poorly named and written so that it is difficult to understand and reuse those sections, or identical sections of code are repeated in several places so that the same maintenance needs to be performed is several places, does that not add to complexity?

There will always be errors in software and therefore depending on the criticality of the error and software, it will need maintaining. Less complex software is easier to maintain because there should be less errors to begin with, and the errors which do exist are easier to locate and fix. Even pre-deployment there may be so many errors and the software may be so complex that fixing them all takes so much time that the software never gets released. Software today is at a crisis level, in the sense that many software projects are too complex to be reliably maintained. Complexity is not a new problem for software developers as described by Brooks.Jr. [4] in 1987 but the measures used to combat the problem in the past are no longer sufficient to manage the increasingly complex software of today. These measures include object-oriented programming and artificial intelligence which are in regular use today, and do improve the situation but only up to a point.

Measuring software complexity is advantageous because if we know how complex an individual piece of software is then we know how reliable and maintainable it is. If we then attempt to reduce the complexity then, through new measurements, we know how successful our attempts were. There are several thoughts on the elements which constitute complexity and of ways to measure for it. For instance [5, 6] show the software metrics which are involved in quantifying complexity. This is not a simple task. McCabe [7] introduces an often used and adapted approach to measurement. It it known as cyclomatic complexity, where a graph theoretic approach shows the number of execution paths in a system. In an object-oriented programming world we should also consider complexity in terms of coupling and cohesion as is discussed in [8] by Darcy et al.

Measuring complexity in autonomic and dynamic software is more of a challenge because the system is not fixed and we may never know at design-time what the changes will be or even when and how often changes will occur. Complexity is however the reason autonomics exists. Autonomic systems combat complexity by reducing the need for human maintenance of a system.

1.1.2 Autonomics

Autonomic computing was first described in [1] by Kephart and Chess and detailed in the IBM manifesto [9] as a way to tackle the growing complexity of modern computing systems. The realisation that human beings cannot hope to effectively manage such large and complex systems themselves has lead to the idea that systems should be able to manage themselves, similar to the way the autonomic nervous system of biological systems is employed to delegate some tasks as background activities. That is tasks which are performed without any conscious thought or apparent effort, such as breathing or fighting disease.

An autonomic system is comprised of four main self-properties. That is, self-configuration, self-healing, self-optimising and self-protecting. They describe the event types which the software is capable of self-managing. For instance, discovery, installation and management of software drivers for a physical device could be considered as self-configuration although varying levels of self-configuration could exist between autonomic systems.

[1] also describes how an autonomic system should be comprised, where each managed element is controlled by an autonomic manager and the relevant environmental context would be continually monitored and analysed. Planning for changes could then be made by consulting a knowledge base of past events and their effects. Managed elements could communicate with each other through their managers. The controller is known as the MAPE controller- Monitor, Analyse, Plan and Execute and is shown in figure 1.1.

1.1.3 Teleo-Reactive programs

T-R programs were designed by Nils Nilsson [2, 10], at Stanford university as a method of autonomous agent control which is well suited for its purpose of guiding an agent to its goal. Traditional languages are less than ideal because of their procedural nature and unwillingness to adapt and change, so much of the agents

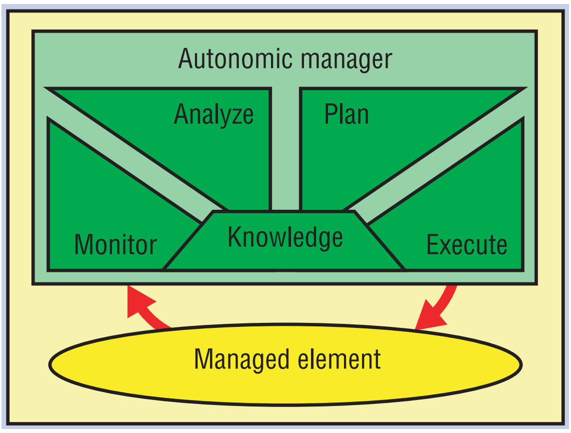


Figure 1.1: Monitor, Analyse, Plan, Execute (MAPE): A managed element governed by an autonomic manager (Originally described in [1])

environment is unknown pre-deployment; an environment which is under constant dynamic change. What was needed was a new paradigm where the environment could be constantly monitored and the agents behaviour would naturally flow with this current environmental context. Therefore a T-R agent chooses the best possible state to be in at that point in time. The program does not follow any predefined pattern and is constantly adapting to the current situation whilst at the same time striving to achieve the set goal.

$$\begin{aligned} Is5000 &\longrightarrow Nil \\ Is1000 &\longrightarrow Add500 \\ Is100 &\longrightarrow Add100 \\ T &\longrightarrow Add10 \end{aligned}$$

Figure 1.2: Simple JTRAF composition test (Counts to 5000)

Figure 1.2 illustrates how this concept works and is explained in more detail in chapter 3. T-R programs are a prioritised production-rule list where conditions can only be true or false and actions contain the code to be executed when the condition is true. Each condition-action pair is referred to as a rule, so for instance ‘Is1000’ is a condition and ‘Add500’ is an action, together they make up a rule. An action executes when it has the highest precedence of any other rule in the program and its condition is true, where higher rules in the program have precedence over lower

is-grabbing	→	*nil*
at-bar-center ∧ *facing-bar*	→	*grab-bar*
on-bar-midline ∧ *facing-bar*	→	*move*
on-bar-midline	→	*rotate*
facing-midline-zone	→	*move*
T	→	*rotate*

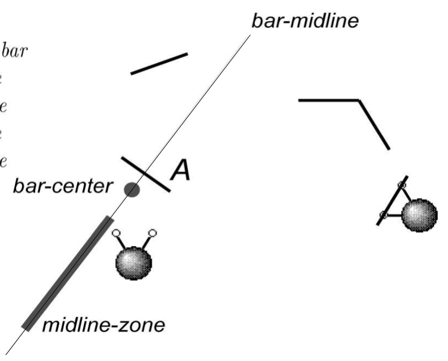


Figure 1.3: Bar Grabbing Robot (from [2])

ones. So for example in the simple counter of figure 1.2 if 'Is5000' were true then its action 'Nil' would execute. If 'Is1000' were also true then 'Nil' would still execute as 'Is5000' has higher priority over 'Is1000'. Each rule works towards its subsequent rule until the highest condition becomes true (the goal condition) at which point the program usually does nothing other than monitor the state of the environment. In this way, a program can move freely between rules and can be either helped or hindered in its progress by external influences. The first condition is often always true, ensuring that if no other condition is true it can always return to this point and proceed again towards its goal.

To better illustrate this, figure 1.3 shows a robot tasked with grabbing the bar at A. This example is taken from Nilssons original paper on T-R [2] and serves as an excellent illustration of the concept.

If no conditions are true then the program must begin at the always true T condition. The action associated is rotate, which the robot does until a higher priority condition becomes true, in this case 'facing-midline-zone' will become true and take priority over lower conditions. The robot then stops rotating and begins to move until 'on-bar-midline' becomes true, at which point it will begin to rotate again until the next condition is true. This continues until the top condition becomes true.

The advantage of this is that if the process is in some way interrupted by external influences, the robot continues at a state in the program which is the closest to the goal and where the condition is true. We do not know how or what interrupted the robot, all we can possibly know from the point of view of the program is that a condition which was once true is now false. Or perhaps a higher priority condition is now true, either way the robot just continues wherever possible at the time.

In this particular program the state is changed if external influences change it. For example, if the robot position or heading is altered suddenly by physically moving the robot to a random position in the test area, then the robot may change to the initial rule state and begin searching for the mid-line zone again. Similarly, if the robot is physically placed on-bar-midline the currently active rule would change to reflect this, it does not matter if this change is a forward or backward step. Although figure 1.3 is an excellent illustration of the T-R concept, this basic program cannot cope with every conceivable situation. For example, what happens when an obstacle is placed between the robot and midline zone? The robot would no longer be able to reach the zone and would just run into the obstacle.

T-R programs can be extended quite easily with the use of sub-T-R programs, and in the same paper from Nilsson [2] this basic program is extended to allow for obstacle avoiding abilities. Using sub-T-R programs for this purpose means that a hierarchy of T-R programs can be added and adapted without the need to change an already working original program. This keeps the programs small and cohesive and even allows for recursive techniques.

1.2 Objectives and Contributions

If an autonomic system is capable of managing itself to some degree then we need verification that the T-R method is a valid way of achieving autonomicity. We need to show whether a system which has been designed in a T-R way displays self-management behaviour. Secondly, if the T-R system is shown to be a valid autonomic solution, then we should ask whether T-R can be improved in some way, to improve reliability, trustworthiness or ease of use; thus improving the potential application and benefits of T-R in autonomics.

Testing this at a quantifiable level is difficult since there is currently no respected measure for how autonomic a system is. If there were, then we could apply the measure to T-R and compare this against other autonomic systems. Producing such a quantitative measure is an area to investigate further in autonomics.

This project asks can T-R be used as an autonomic technique and if so, what can be improved in order to make it easier to use and a better fit into the target domain of autonomics?

In autonomic computing we aim to reduce the maintenance complexity to a manageable level by allowing the software to manage itself to some degree, i.e. an autonomic system, once deployed, should require very little input from a human to maintain its stability and usability because these tasks should be handled by the software itself to some degree. The T-R system seems appropriate for autonomic computing because of its focus on conditions and goals. If any condition changes from true to false then this could be normal behaviour or could be caused by many possible types of error, even errors which are unexpected. The T-R program should take appropriate action for this condition change. In a standard procedural type language an unexpected error may cause the system to fail and therefore a human would be expected to write some extra code to prevent this error (now known and possible) from occurring or reduce its impact so that it no longer causes the same system failure.

Once an error has been dealt with by human external intervention (maintenance) there could still be further problems and so maintenance is required almost indefinitely. A condition in a T-R program on the other hand could encompass many errors, known and unknown, and even though the T-R program may not ‘fix’ the error directly, it should prevent unrecoverable problems and reduce the need for human maintenance in this way.

The T-R paradigm could be considered an autonomic technique because it is a more natural and simpler way of approaching a problem area and reducing complexity through reducing the workload of human developers. As such it could be a more viable option than some other autonomic techniques. The paradigm is a very different approach to other methods, it doesn't try to attempt the impossible of fixing every conceivable problem. Instead the T-R paradigm works towards a goal without ever knowing any specific errors which can occur, whilst still safeguarding against the errors.

Applying a T-R approach to autonomics is one which is quite different to previous ideas, relying on a programming paradigm and not on an autonomic architecture or other approach to achieve results. Some of these currently existing techniques are relatively heavyweight and complex themselves, therefore this work implicitly asks the question of the minimum cost to achieve autonomicity as there is a danger of producing a complex framework whilst at the same time attempting to reduce complexity. This was termed 'complexity tail chasing' by Richard Anthony in [11].

The initial research identified some incorrect direction from others. Whilst there has been some good research, it has not always been directed towards the intended goals of autonomics. This shows a lack of definition of terms and ambiguity in targets. Although this project will not focus on defining terms or removing ambiguity, it will attempt to highlight the correct approach.

Large projects are inevitably affected by software complexity issues. Often, software developers cannot afford to continually maintain their products so a system which can reliably manage itself after deployment will have obvious benefits.

Given the initial study, the specific *research questions* addressed by this work are:

- To what extent can Teleo-Reactive programming techniques (T-R) be used to construct autonomic management systems and what are the benefits of doing so?
- What are the difficulties in T-R programming and how can they be reduced?

- What improvements can be made to extend the benefits or increase the ease of adopting T-R programs?

These questions are explained, detailed and related to the method in chapter 3.

CHAPTER 2

LITERATURE REVIEW

The project aims to investigate and apply T-R programming techniques within the autonomics domain. This review of the literature therefore, extensively investigates the current state and thinking in autonomics and T-R work, including the different approaches and architectures that have been applied and the results from this work.

The previous work on T-R programming has largely focused on the control of autonomous agents, since the technique was developed for use in robotics. However, much of this work could be applicable to higher level and autonomic applications, and it is this idea we investigate. It has also been observed that goal-oriented software design has an inherently close relationship with T-R programming techniques since it could easily be argued that the highest condition of a T-R program is the main program goal and lower conditions are sub-goals. The current uses and state of this domain is also investigated here.

T-R programming offers a natural fit with many artificial intelligence techniques and there has been much work published towards adapting these ideas for learning and dynamic changes. The literature review will therefore cover some dynamic adaptation techniques. Techniques used in conjunction with T-R systems are shown as well as techniques applied in higher level and autonomic systems.

2.1 Dynamic architectures, techniques and middleware

'Autonomics' and 'self-management' are generally interchangeable terms and if a self-managing system is capable of altering itself at runtime to adapt to changes to the environment or to unexpected events, then it is reasonable to assume that any autonomic computing system must be capable of some form of dynamic adaptation. This could be a human controlled change such as replacing one policy script with another at runtime as this work on AGILE policies shows [12, 13, 14, 15, 16] in order to fix a design problem or make an improvement. Or it could be an architecture which supports on-line change where the system automatically makes adaptations to itself at runtime, [17, 18, 19].

2.1.1 Dynamic architectures

The work on AGILE Lite and the architecture to support AGILE policies is described here [12, 13, 14, 15, 16]. In this architecture, the system is operated as normal and AGILE policies can be inserted at various points, known as decision points, where a change is required. The system allows policies to be written off-line and integrated when discovered. Several components are used, including a context manager which reads information from a dynamically changing environment and feeds this to the corresponding policy. Another component is the repository service which manages the policies used for each decision point. It should allow an error producing policy to be 'rolled back' to a previously known working policy.

Reflection gives code the ability to alter itself at run-time to any extent as stated in [20] and shown by Dowling et al. in [21]. It gives the ability to change an object's interface or even modify, add or remove internal workings of an object. Reflection is used in these systems [17, 22, 23] which has a large potential for dynamic environments where the state of system at any point is often unknown at design time and unpredictable changes need to be made to a system which cannot be halted. Reflection is very powerful in the sense that almost any software related task is achievable through reflection where it may have been impossible without. There are however some problems associated with reflection, such as slow invocation time, since methods are often discovered and invoked from a disc rather than through compiled code. It is often a more convoluted approach to method invocation and it does offer a way to circumvent good coding practices such as encapsulation, so could easily add to complexity. For example, variables which were designed to stay hidden and marked as ‘private’ could still be open to inspection and use by reflective code.

Aspect Oriented Programming (AOP) is described by McKinley et al. in [24] as a valid way to implement adaptable behaviour. AOP is a large paradigm shift from other more established programming techniques. With AOP, concerns are separated into *cross-cutting concerns* such as security or logging. The *aspects* implementing the concerns are developed separately from the main system and are woven into the code at a later date. For example a security aspect can be applied to a piece of code containing sensitive data to be sent over a network. The aspect is introduced at a specific point in the code and at any other point where it is needed. The aspects, when woven with the main code, form the complete application. The aspects can then be changed independently of the main code. AOP could also be used to create autonomic code from legacy code by adding new aspects into the existing code. This is a goal of the model-driven autonomic architecture in [25] in which a policy managed architecture is used to generate autonomic systems from non-autonomic components.

Another component based reconfigurable architecture is described by Kramer and Magee [26]. It uses a similar 3 level goal management, change management and component control approach found in some robotic systems. Components in the component layer which require reconfiguring must do so in a manner which adheres to application consistency and does not introduce undesirable transient behaviour. The change management layer ensures this by decentralising the view of the system state whilst tolerating inconsistencies and still converging to a stable state. To ensure changes are consistent, the goal management layer is relied on. The authors clearly advocate an architectural approach to autonomic solutions but it is unclear whether the technique described has yet been implemented. As such it is difficult to ascertain its viability.

A layered architecture of context-aware software infrastructure, as well as layered structure of context information management, is proposed by Henricksen and Indulska [27]. This work also presents commonly used context modelling and context abstraction techniques that support the decision making process involved in mapping of context information to appropriate application behaviours. The adaptation layer uses three repositories: situation repository, preference repository and trigger repository and dynamically provides the application layer with appropriate services from the lower layers.

Henricksen and Indulska also developed a graphical Context Modelling Language (CML) tool for designers to specify required context for context-aware applications. It provides modelling constructs for describing types of information, their classifications (sensed, static, profiled or derived), relevant quality meta-data, and dependencies amongst different types of information.

Dashofy et al. [28] use a large number of connected components to support adaptation. The required changes are simulated to verify the proposed changes. If successful, the changes are merged using an architectural 'diff' tool. A similar technique of simulating proposed changes is presented in Butler et al. [29].

Garlan and Schmerl [30] also use a complex model of interconnected components to monitor, analyse, perform adaptation etc. The model is 'externalized' from the running system, providing a way to monitor and understand the system in a high level way. This is quite similar to many meta-modelling approaches where the level of abstraction allows changes to be made without the need to know the precise details of the program code.

In contrast to these systems, Hassan et al. [31] use case and rule based reasoning to make a correct decision about how to proceed in the case of problem events. The solution is stored or retrieved from the case-based-reasoning (CBR) system if it has previously been encountered, or contacting the domain expert if not.

2.1.2 Middleware

Huebscher and McCann [32] describe a middleware architecture consisting of sensors which provide their information to context providers and are managed by context services. The multiple context services (equivalent to the context managers of [12, 13, 14]) choose the best context provider based on the notion of "context provider quality".

This level of abstraction provides robustness and scalability but requires additional services for the application to configure the most appropriate context provider. In this design the context providers are required to periodically inform a directory service that it is still alive.

Another notion considered by Huebscher and McCann in [32] is "trustworthiness". Here, applications are required to submit a complaint or praise for their associated context providers. This increases or decreases levels of trust for the CP's and can be considered in the utility function by the context service.

AOP has been used in adaptive middleware as Gilani et al. describes [33]. They work on dynamic weavers in order to implement run-time changeable code on embedded systems. Like reflection, AOP can be very powerful and can also be used to circumvent measures designed to prevent problems occurring. Measures such as encapsulation designed to prevent interference from external sources could be rendered ineffective.

The RUNES (Reconfigurable Ubiquitous Networked Embedded Systems) project is an architecture for networked embedded systems. It includes dedicated radio layers, networks, middleware and simulation and verification tools. A middleware platform, which makes extensive use of the RUNES architecture, is described by

Costa et al. [34]. The component based platform uses well-defined interfaces to decouple the components. This allows platform developers to deploy different variants of the same component whilst still allowing dynamic reconfiguration of component instances and their interconnections. This provides support for dynamic adaptation to changing conditions. A fundamental requirement in context-aware scenarios is that it can dynamically adapt its behaviour which is a requirement met by this system. More sophisticated behaviours and functionalities are achieved by collecting components within 'capsules'. Access to these functionalities is possible through the capsule API.

A Service-Oriented Context-Aware Middleware (SOCAM) as well as a formal context model used to address issues including context reasoning, context classification, and dependency are presented by Gu et al. [35]. The architecture aims to provide an efficient infrastructure to support building context-aware services that are assumed to make use of different levels of context and adapt the way they behave according to the current context. In this approach context providers abstract context information from low level sensors to higher level context manipulators. The context interpreter is a component that is responsible for context processing using logic reasoning. Context services at the highest end of the model perform operations on the various levels of context.

2.1.3 Conclusion

It is clear from these designs that an architecture to enable an autonomic system will be quite complex itself, using many complex components and connections to support the ideas. A complex architecture is both positive and negative. On one hand, the complexity of the architecture can be hidden from the utilising autonomic system itself. If the developer is involved in just the main system design then the autonomic architecture need not be their concern as self-* properties should be present here. However, the chances of failure of the architecture increases as its complexity increases.

This leads to the question as to whether or not a complex architecture is the best approach for the development of autonomic systems and whether there is a simpler method to achieve the desired result.

2.2 Autonomics Objectives and Current Progress

Writing software to meet the demands of users and clients is becoming more and more difficult due largely to the accompanying maintenance cost. This cost arises due to the structural and behavioural complexity of software and the scalability of systems. The functionality embedded into a modern system has increased with the demand for it but the capacity for maintaining the system is unable to keep pace with this demand. This is not a new problem, and has been observed and predicted previously. At the time when [4] was written there were promises of technologies which may hold an answer to this problem. These 'solutions' included AI techniques and Object-Oriented programming, many of which are the main backbone of software engineering today. The problem is that we have now reached the software complexity limit even using these technologies and we now have the same problem as before.

Autonomic computing [1, 9] is a branch of computing which acknowledges the fact that the larger a system is, the more unmaintainable by humans it becomes. An autonomic system should be able to manage itself to a degree, thus reducing the need for human maintenance and administration. The idea is inspired by biology, specifically the autonomic nervous system, where tasks such as breathing and repairing of damaged tissue, bone and cells are performed autonomously, that is without the need to consciously think about it. A software system should be able to do the same. i.e. perform some tasks which require little human attention, autonomously, thus reducing the problem of managing and maintaining a large, complex software system.

In autonomic systems much of the complexity is dealt with by the system itself and not the human developers, thereby reducing an impossible maintenance task to a more manageable one. Self-managing systems consist of one to many self-properties, the most widely used and referenced are the four first described in [1], namely self-configuring, self-healing, self-optimising and self-protecting, or 'self-CHOP'. Effectively, systems exhibiting these behaviours react to and deal with changes. For example, a self-configuring system might detect that a new printer has been attached to the computer, find the correct driver, install it and add it to the list of available printers. Whereas a self-healing system might detect an error in a piece of code, diagnose it, apply a fix if one is known or roll back the software to a stage before the error occurred. All this whilst limiting the need for human involvement.

An autonomic system can be viewed as a collection of inter-operating elements within autonomic managers, see figure 1.1. This manager both controls the component behaviour and its interaction with other elements. Figure 1.1 shows the monitor-analyse-plan-execute cycle which is present in autonomic systems, directly or indirectly.

Sterritt et al. [36] provide a concise introduction to autonomics and Huebscdher and McCann [37] provide a more recent survey on the field. Thaddeus O. Eze et al. provide a very comprehensive review of the state of autonomics in [38] and Jeffrey Kephart's keynote at ICAC 2011 [39] provides a similar description of the current state of autonomic research, a decade from its first seminal paper [1].

2.2.1 Current Techniques

An autonomic solution need not use a specific technique but should be required to complete its goals whilst reducing human intervention. For many deployed autonomic systems the solution is aimed specifically at an individual problem rather than a generic autonomic technique able to be adopted in many situations. These autonomic systems, however narrow their view have been hugely beneficial in testing the validity of autonomic computing and to discover an appropriate approach. The following section discusses some of those techniques.

Policy-based techniques

Policy-based computing techniques abstract operations and behaviours into a higher level, more constrained and convenient representation (policies). A key aspect of this is separation of the control logic from the underlying implementation mechanisms, which in turn has the advantage of giving the policy designer the right to only make changes to the system where the system designer wants changes made. A policy script should describe what is required and not specifically how to do it. A policy script should be easy to create with shorter, more natural language constructs used (more human readable) in comparison with the more technical language the system is based in.

Policy technologies usually provide general guiding strategies to a system. They inform the system how to perform the tasks without replacing the underlying code. Ponder [40] permits run-time changeable security policies and has a very feature rich and extensible grammar. It is a good choice for using in self-adapting policy-based security software but may not be as well suited for applications without high security concerns. Ponder has been used in [41] for the management of differentiated services networks.

AGILE [42] is the policy language used in DySCAS [43]; a European project concerned with the development of dynamic processes of non-critical tasks within and around a vehicle. An AGILE policy script can be loaded into an application at run-time with the purpose of changing the behaviour of the application at the point where the script is inserted. The scripts are loaded and processed by an AGILE library instance. The DySCAS middleware supports run-time adaptability through the use of AGILE-Lite; a lightweight, embedded version of AGILE [12, 44, 45].

Although powerful, AGILE policies at the same time can define functionality at a high level so that developers can focus on the intended business logic and need not be experts in concepts such as autonomics and policy-based configuration. A policy can be a statement of intent, without having to describe exactly how the behaviour is achieved (since the lower-level mechanisms are pre-built). An AGILE policy editing tool further simplifies the task of preparing policy scripts; making script editing less problematic and less error-prone. For more information on AGILE policies see [46, 11].

Using AGILE policies in DySCAS means it is possible to defer addressing issues to a time when it is needs addressing. The system may start with a simple policy for control of a particular service then a new policy can be created which takes into consideration the additional context and is inserted into the service at run-time [42].

Rei [47] policy language consists of action and condition pairs whereby actions are performed depending on the evaluation of its condition, i.e if and only if the condition is true, does the action evaluate. Within Rei, prohibitations, obligations and dispensations can be assigned to agents. Actions are constructed using a name, target objects, pre-conditions and the effect of the action. All of which leads to a policy language which is very powerful and can be used across domains but comes with a steep learning curve.

Imperius [48] is very similar in that it uses condition-action pairs within a policy language. Policies execute next to a system and can inspect the context of the classes. This context is often used as part of the condition for deciding if the action should be executed.

Utility functions

Utility functions are a way of dynamically selecting the most appropriate of several options based on the contextual ‘value’ of these options. Utility functions are available in AGILE policies [11] allowing system designers to implement some complex behaviour with the use of multiple terms and weights beyond simple true or false rules.

Kephart and Das [49] argue the case for the use of utility functions in autonomic systems saying the "they provide a principled basis for automated agents to make rational decisions". They compare three types of policy as part of the case, action, goal and utility function policies. They say that utility function policies are more appropriate for autonomic computing than action policies because they focus on the desired state.

It could be argued that a system goal is in effect a desired state. But the main advantage of utility function over goal policies is the ability for the system to compute its own, most desired state based on scalar values rather having the policy author specify the most desired state. Therefore, where multiple goals would conflict, utility functions would take a rational decision automatically.

Das et al. [50] use utility functions to attempt to cool a datacentre environment in an energy efficient manner. They have produced some positive results in regards to the experiments they performed. A device which they refer to as a 'snorkel' is used to "channel cold air to the inlet of equipment mounted high off the floor" as well as two computer room air conditioners are used to aid cooling.

The authors say that utility functions can be used to quantify and manage trade-offs between competing goals such as performance and energy consumption. They conclude by saying the datacentre energy consumption is reduced by 12% using their utility function method (nearly 14% when using snorkels).

Other cases where utility functions are used in a datacentre environment to a positive effect are shown in Tesauro et al. [51] and Walsh et al. [52].

Machine Learning

Machine learning is a technique in which the weight of successful strategies is automatically reinforced to increase the chance of using the most appropriate strategy in future similar occurrences. Using reinforcement learning to implement a control mechanism that provisions servers and places applications in a 'cloud' computing environment, whilst meeting performance requirements and minimizing costs, is the theme of Li and Venugopal [53]. This is split into two sub-challenges. Firstly, to determine the smallest number of servers to satisfy the resource requirements. Secondly, to distribute the application amongst servers in order to meet response times and availability requirements.

The results from experiments of Li and Venugopal [53] show that the system coped well with some unpredictable request data after an initial transient period where the system was 'still learning'. This does however put forward the question of just how beneficial reinforcement learning could possibly be if request data was truly unpredictable. There could be a limited pattern to spot a trend and other, less complex techniques might perform equally as well.

A self-managing system and in particular, a self-healing system should be able to make improvements to itself to better handle the same and similar errors in the future. The 'self' part indicates that these improvements must be made independent of human guidance and artificial neural networks (ANN) as described by Haykin [54] provide one method of unsupervised learning. Here, the ANN attempts to mimic the brain functions where input and output nodes represent neurons and the connections represent synapses. The actual output nodes are compared with the ideal and the error between the two is calculated. The values of the nodes are then worked back through and adjusted to improve the output. The process is then repeated, possibly millions of times, with small improvements made each time.

Al-Zawi et al. [55] say that supervised learning of autonomic systems is not feasible. In this paper a pipelined recurrent neural network (PRNN) algorithm is used to implement self-healing in a real-time environment, i.e. the running system cannot be interrupted and monitoring, planning and configuring must be made outside the running system.

It could also be argued that all autonomic systems are closed control loops as all self-* properties need to examine the current state of the system and pass the information to the system as feedback in order to 'self-adapt'. The same argument could be applied to the MAPE controller, i.e. though most developers do not strictly adhere to it, but all autonomic systems contain the elements of MAPE, and MAPE is in fact a closed control loop. Gaudin et al. [56] take control-theory further and base a self-healing system on it. Here Supervisory Control Theory on Discrete Event Systems is used to automatically synthesize a new supervisor that disables the system functionalities that led to an exception so that the same sequence of events do not re-occur.

A utility computing environment is one in which computer resources such as memory and processing speed is treated as a utility in the same way as water and electricity. These resources are given a cost to use and therefore the problem in this environment is when and whether a particular investment is worthwhile.

In CARVE (Cognitive Agent for Resource Value Estimation) [57] the system value of having more or less resources is predicted. This involves the training of CARVE's models, evaluation of on-line machine state and performing any necessary reallocation. The experiments show the effectiveness of the system in comparison to other static systems.

Andrzejak et al. [58] target a very similar problem, i.e. predicting resource usage for the operation of a datacentre. They too target utility computing as a problem area. The authors say that in such dynamic environments it can be very difficult to predict resource usage with limited training data, and propose a genetic algorithm / fuzzy logic approach to predicting where there is scarce historical data available.

The experimental results show that their method does indeed provide much smaller error predicting resource usage over other linear models especially when limited training data is available.

These and other methods for implementing autonomic systems are surveyed and discussed by Khalid et al. [59].

Kalman Filters

A Kalman filter is an algorithm which allows system models to predict the state of some value where the sensors used are unreliable and/or produce a certain degree of noise. Since real world sensors almost always produce noise, especially when the values being measured are of analogue nature, this can be an extremely useful technique in computing. The basic formula governing a Kalman filter is thus:

$$x_{t+1} = x_t + V_t$$

where the state of the future system x_{t+1} depends on the current system x_t plus some noise V_t. The error between the predicted state and the real state can be reduced over time and used to improve the output of the filter.

For autonomic computing in particular this could have potential benefits as a system that can make accurate predictions of its state can make more accurate and trusted changes to itself. Hence the system should have a high degree of self-managability which reduces further the need for human involvement at the maintenance stage.

Capra and Musolesi [60] apply Kalman filters in a pervasive computing environment. They use the filter to assess the trustworthiness of various competing services. Attributes pertaining to the quality of the service are measured and compared to promised values and these values are used to train the Kalman filter and assess the trustworthiness of the services.

Kalyvianaki et al. [61] focus on a virtualised datacentre environment, an area which has become an important and popular research area recently. In these infrastructures one physical server can contain several virtual servers thus the physical resources are shared between them. Kalman filters are used here to better allocate CPU resources to the service which requires it by monitoring usage and using the output of the filter to guide new allocations.

As mentioned, virtualised environments are an important area of research, not least because of their widespread use in almost all datacentres. The problem of where to dynamically allocate resources however may or may not be a big problem if there are enough resources to share statically or the criticality of the server and service is low and the need for efficient resource allocation is also low.

Zheng et al. [62] use Kalman filters to track a time-varying parameter in a layered queueing model within a service centre. The graphs and experiments show the effectiveness of the approach. The mechanism used for monitoring and changing the application, as well as construction of an effective model, could be complex however and therefore not a practical approach for many systems.

Kalman filters are similar to state space observers where the variables of a system are unknown but the state can be reconstructed by observing the output of a system and performing algorithms on the output. Pelc et al. [63] apply policy supervisors (which can themselves be dynamically changed at run-time) to select the most appropriate state observer from a set based on the current context information. A variety of comprehensive simulations are performed to show the quality of the observer with regards to the error ε between the real system state $x(t)$ and the estimated system state $\bar{x}(t)$:

$$\varepsilon = x(t) - \bar{x}(t)$$

The simulations show:

- the performance of the control system,
- the quality of state estimate,
- the stability of the closed-loop control system

2.2.2 Challenges

The majority of systems in use or in production today target a fixed goal and are set on achieving that goal. Despite past experience, designers either do not expect that errors might occur along that path, or do expect them and fix and maintain when they occur. The problem is that this list of errors is infinite, may never be completely solved and may cause the system to never complete its aims and objectives. This leads to an ongoing maintenance cost, a cost which is impossible to ever satisfy.

An autonomic system must therefore adapt to unexpected events and errors and plan to deal with the new state itself, therefore reaching its goal despite the problems. This is one of the fundamental aims of any autonomic system, some of the current methods for achieving this have already been discussed.

Further to the fundamental challenge of designing an autonomic system, Jeffrey Kephart in [64] discusses some of the foreseeable challenges involved. Here, interoperability is regarded as one of the major challenges. If multiple autonomic managers exist, then how can they successfully communicate with one another without one manager having a detrimental effect on another? One high level approach mentioned would be for elements to include "models that map potential actions into probable outcomes so that they can make decisions about the right action or action sequence".

Kephart concludes with a vision of a open autonomic platform to which contributions can be made from the autonomic community similar to the PlanetLab effort (www.planet-lab.org). This is an idea which could be worth pursuing as it would probably push the autonomic effort forward, plus it would be likely to enable a much higher degree of standardisation between autonomic elements which in turn would increase interoperability levels.

Measuring and effectiveness

An important consideration in autonomics is a way to measure a system for levels of autonomicity. Is it possible to obtain quantifiable results for the system as a whole or individual self-properties?

As we have seen, there are a vast array of proposed autonomic systems using a variety of methods employed in different problem domains. But suppose a client needs an autonomic system to solve a particular problem and there are already a selection of systems which are desirable to them. The client does not want to propose a new solution but make use of an existing one. How can the client know which system is best for their needs? If each system could be quantifiably measured then the client would be in a better position to make a choice as to which to employ. Perhaps the client's system needs a high level of self-configurability, but is not too concerned with other self-properties, then one system might be better suited than others because it scores well on that particular self-property but not so well on others. In short, measuring allows for comparison with other systems.

Another benefit of measuring autonomic systems is to aid developers in improving existing systems. Measurements could be run after a particular improvement which has been designed to increase self-optimisation for example. The new measurments would be able to communicate the effectiveness of the impovement. This also gives developers an effective way to back up their claims that a system is particularly effective at self-healing for example.

Huebscher and McCann argue the case for the need to measure autonomic systems in [65]. They describe possible metrics that could be used for measuring, including quality of service, failure avoidance and adaptivity metrics. They argue that the metrics should be part of a benchmarking tool.

Brown and Redlin [66] obviously see the need for a benchmarking tool as well since they have built and use one to measure the effectiveness of self-healing in an autonomic system. They use the IBM Autonomic Computing Maturity Model [67] as a basis for quantifying the results of the test. The tool injects faults into the system under stress and a measure of how effectively the system heals itself in response to the injected faults is taken. It also measures how autonomic the healing response is in accordance with the maturity model.

Trust, Validation and Verification

Autonomic and dynamic systems in general adapt to changes at run-time and perform operations which might not exist at the time of design. Therefore it is often impossible to know the behaviour of the system at any point in time. Clients often need to trust that a system will always behave within certain boundaries, especially in the case of safety-critical systems, so some form of system behaviour guarantee is often imperative. Furthermore, an autonomic system could change a section of one element which might not only cause problems for that element but also for other elements which rely on it.

Validation and Verification have two distinct meanings in software as defined by Adrion et al. [68]. These definitions are:

*"**Validation.** Determination of the correctness of the final program or software produced from a development project with respect to the user needs and requirements. Validation is usually accomplished by verifying each stage of the software development life cycle."*

*"**Verification.** In general, the demonstration of consistency, completeness, and correctness of the software at each stage and between each stage of the development life cycle."*

By using these definitions and using our own interpretations, Validation is the correctness of software according to the clients' intentions. Verification is the correctness of the program in general. Does it contain faults or fail at any point causing incompleteness?

It is difficult to totally verify that software is completely correct because we cannot know every possible configuration and thus we cannot verify the unknown variations. Through using various verification techniques we can however improve robustness and even prove the behaviour of software in certain situations.

The problem of validating a system often comes from the possible ambiguity of the intentions for the system as a whole. If there is any ambiguity then several developers could interpret those intentions a multitude of different ways, each way is correct in the way that it fully conforms with the original design brief, yet it does not match the original picture of the system. Even without ambiguity it may be realised that the original intentions were flawed as the development process becomes more complete and that they need to be revised.

If no compiler or interpreter can be used in a system then code which is added or changed during run-time needs to be semantically validated to ensure the safe running of the system. Semantic validation is necessary for the run-time changeable “unknown” code and also for the entire system. Kamel and Leue [69] describe a visual editor (VIP) which is used as an interface for a compiler of Promela code (the input language for the model checker Spin). The visual model acts as a formal description which can check the semantics at the point of system design. This means mistakes in the intentional meaning of the designer are exposed and reduced. A visual editor can be beneficial in validating and specifying high-level intentions.

Model checkers are often used to verify system requirements and have been shown to verify policies, as demonstrated by Kikuchi et al. [70]. Spin is again used as part of a framework built using the system model and policy. The result of the model checking procedure must satisfy constraints and final states provided by the policy.

Petri net models Desel et al. [71] and Huang and Kirchner [72] are an effective method of Validation and Verification. The models constructed offer a graphical representation and a mathematical description of the system.

The problem of verifying adaptations of a system has been attempted by Zhang and Cheng [73]. Petri net models are constructed of the system before and after adaptation. These can then be checked for unacceptable state changes. They use global invariants to verify the system, i.e. states which must not change after adaptation. This assumes that the system will continue to function as expected as long as the global invariants remain satisfied.

Several other types of verification are also available as mentioned by Wolf and Holvoet [74], some of which are described here:

Unit and Integration Testing - This method involves testing individual units of code with test code to verify that the individual parts are correct. This gives the developer confidence that the tested parts will work as they were intended to work. Integration testing verifies the group of units to ensure that the group works when all the individual parts are collected. The initial overhead of writing test code proves worthwhile as it produces very few errors to return to and fix.

Formal Proof - This is concerned with a mathematically provided description of the system in question. Once this model is obtained it can be used as a basis for proof of system behaviour and to ensure errors and faults are significantly reduced. Formal proof is generally not perceived as necessary in all but the most safety critical of systems. This is partly due to the overhead involved and partly due, to the perhaps wrong assumption, that the currently in-place methods of verification are ‘enough’ to ensure safe and correct running of the system.

Statistical Experimental Verification - This method involves running experiments and processing the results using statistical techniques. The system behaviour can be represented by a collection of the experiments which can then be used to verify the conformance with the system requirements.

2.2.3 Conclusion

The argument that a system can never be truly optimal and error free is backed up by Lightstone [75], who claims that the 3 popular middleware products under test had more possible configurations than there are atoms in the universe, taking into account the number and range of parameters which are possible in those products. Clearly, the task of maintaining these systems is far beyond the capability of human beings and clearly there is a definite problem which urgently needs addressing.

Throughout this research we have struggled to define the term ‘Autonomics’. In particular, the difference between an autonomic system and an autonomous one. The Oxford dictionary defines the word autonomous as ‘having the freedom to act independently’ i.e. a software system, once deployed, which has the ability to act independently of human input in order to complete its task is autonomous. For instance, a robot exploring the surface of a distant planet must be able to perform many, if not all tasks independently of human instruction because the delay between signals being sent and received would be too great to allow for remote human control assuming the human would be on earth. The position and circumstance for the robot may be several minutes old by the time it is known about on earth and therefore the human operator would be trying to correct an issue that the robot has long since progressed from and is now encountering another obstacle which the human operator will not know about for several minutes.

An autonomic system is difficult to differentiate from an autonomous one because it seems to be attempting to achieve similar objectives, namely, to automate tasks and therefore reduce human input. Most dictionary definitions of autonomic refer to the autonomic nervous system of humans and other physical bodies whilst there is no specific definition for autonomic computing. For example, the Oxford dictionary defines autonomic as ‘involuntary or unconscious; relating to the autonomic nervous system’. The terms autonomics and self-managing can be used interchangeably however and this term has a more obvious definition. Self-managing software is software which has the ability to at least in part manage itself. This seems synonymous with the term used for autonomous systems, i.e. software which can manage itself and software which can act independently means the same, does it not?

If we look at the original autonomic proposal in [1] the authors envisioned autonomic software elements controlled by MAPE controllers (see figure 1.1). The controller consists of elements which govern the monitoring and analysis of the software system, as well as planning and execution control of changes. Autonomic managers also add and remove information in a knowledge base, which can then be used for future decision making. The original autonomic software systems stuck closely to this design of the autonomic manager but newer systems use different techniques as proposals or implementations of autonomic systems although it could be reasonably argued that any autonomic system could be mapped onto each control element of an autonomic manager.

Does this mean that all autonomic systems must use a MAPE controller? If we claimed this we could equally argue that all autonomous systems could be mapped onto a MAPE controller in much the same way as all autonomic systems. It is intuitive that a self-managing or autonomous system must contain many or all the elements described in the autonomic manager. Also, if there was a system capable of self-management, but could not be easily mapped to an autonomic manager, could we then dismiss its autonomic capabilities? If a system displays autonomic capabilities then it would have to be classed as autonomic no matter which methods are used. In other words, if two competing autonomic systems complete the same goal and both systems reduce the human element to the same degree, but only the first system based its architecture on MAPE, then does this mean only the first system is autonomic?

So what is the difference between an autonomic and autonomous system, and why is the answer important? To answer the second question first, if we cannot find a difference between the two then they must have the same meaning and there is no reason to have two distinct research fields. It also means that we can classify these types of systems more precisely with a distinct boundary between the two.

To define these systems then, we need to look, not at the methods used in the system but at the reasons for automating tasks. An autonomous system is one which automates its tasks because human involvement is impossible or highly undesirable such as the 'planet exploring robot' example used earlier or the flight stabilising system used in several fighter jet aircraft, Cortellessa et al. [76] and Miller et al. [77]. An autonomic system automates its tasks in order to reduce complexity and is often a replacement for systems which usually refer errors, configuration tasks etc. to a human, e.g. an autonomic datacentre manager or autonomic power/performance management system. Autonomous systems are usually based in robotic systems whereas autonomic systems are usually much higher level even though there is no discernible difference in the methods of the two?

2.3 Requirements and Goal-Oriented Engineering

In our framework and in T-R programs in general, the focus is on *goals* as the most influential part of a program. Maintaining focus on these high-level aims is essential in delivering a valid product.

The i* framework, Yu [78] is used extensively in the requirements engineering domain, either on its own or as the basis for an extension such as Tropos, described by Bresciani et al. [79] and Castro et al. [80] or the Goal-Risk framework, Asnar et al. [81]. i* is a modelling and analysis technique used in early requirements stages where stakeholders are modelled as actors in the system and their interdependencies are described. Where earlier frameworks are often used to model *what* and *how* questions, i* also models the *why* question as this is also seen as a fundamental query in requirements analysis.

Each goal in i* is refined into subgoals, where positive and negative contributions between these goals are shown. The resulting model shows alternative ways of satisfying top-level goals.

Goal-oriented requirements analysis and reasoning is the main subject of the work by Giorgini et al. [82] who use the Tropos methodology [79, 80] to make goal analysis more complete, developing a formal model for this aim. The authors have developed a goal reasoning tool, which allows algorithms for forward and backward reasoning to be run on goal models. The backward reasoning in Tropos is used to analyse the goal models to find the minimum cost goal that could guarantee the achievement of top level goals. The forward reasoning is employed to evaluate the impact of adopting the approaches with respect to *softgoals*. *Softgoals* in Tropos and indeed, in many other goal-oriented systems, mean non-functional goals such as *Are customers happy?*.

Van Lamsweerde [83] argues the case for widespread use of Goal Oriented Requirements Engineering (GORE). He argues that it makes sense that the creation of software should be directed towards what the user wants to get from it, i.e. the goal, and that these goals should drive the requirements. The author shows evidence of several successful projects which use GORE and also argues that tool support should be integrated into GORE development.

Amyot et al. [84] also champion GORE as a way forward in software development. They give a detailed account of the relationship between goals in GORE models and use several examples to illustrate their point. For example, the relationship between one goal and another goal or softgoal can have several satisfactory levels and contribution types, including satisfied, none, conflict, denied, some positive, some negative, to name just a few.

Van Lamsweerde [85, 83] places a high degree of importance on high level goals, with the focus on functional and non-functional goals, such as performance and quality of service. The authors of [85] also focus on how to generate these goals in the first place, saying that many goals can be obtained simply by asking HOW and WHY questions to obtain parent and sub-goals. Simple Use Case diagrams described in Fowler [86] are a good way to obtain and focus on initial goals.

Asnar et al. [81] propose a framework where risk analysis is integrated into the early requirements engineering process. They argue in many software project cases, that risk mitigation has the undesirable effect of requiring a revision of the software design and of the software requirements. By shifting risk analysis to the early requirements stage this is a reduced possibility.

Sutcliffe et al. [87] discuss the use of some HCI approaches to requirements engineering and applies these methods to the development of the ADVISES tool. The aim was to develop "user-centered RE techniques and processes, to specify functionality for multiple-user communities" and investigate the integration of HCI techniques as part of a project solution and RE process.

2.4 Teleo-Reactive Programming

T-R programs developed by Nilsson [2] are designed for autonomous control of mobile agents. T-R programs continually accept feedback from the environment, performing actions based on this current state. A T-R program is structured with a hierarchical list of condition and action pairs with each action fulfilling or partly fulfilling the condition of immediately higher precedence. An action will end execution if it ceases to be the highest true condition, either because the action has fulfilled the next condition or some other circumstance has caused this case.

Nilsson develops a triple-tower architecture in [3] which consists of perception, model and action towers. The model of the environment is updated through sensory systems, creating primitive perceptions in the model tower. These perceptions are both added to and deleted by evaluation from the perception rules. The action tower consists of T-R rule lists which essentially perform the actions based on current perceptions in the model tower. The actions can of course affect the environment and contribute to the perception tower, creating a feedback loop. The architecture is shown in figure 2.1.

This allows conditions to be continuously evaluated, new perceptions conceived and actions to be re-ordered according to current state. A Java applet providing a demonstration of this block stacking example is available at [10].

Hayes [88] has produced a reasoning framework, supporting verification of T-R programs. This is especially important in the case of safety and mission-critical software and increases confidence in the developed system. It is likely that the T-R program itself contains logical errors causing the program to never progress past a particular state. Being able to reason over a T-R program would have benefits then.

Basing their approach on T-R programs, Gordon and Logan [89] have designed GRUE (a Grue is a monster from an early adventure computer game, Zork), an architecture for controlling game characters (agents). With GRUE the agents are able to react to competing goals with conflicting requirements. GRUE is able to

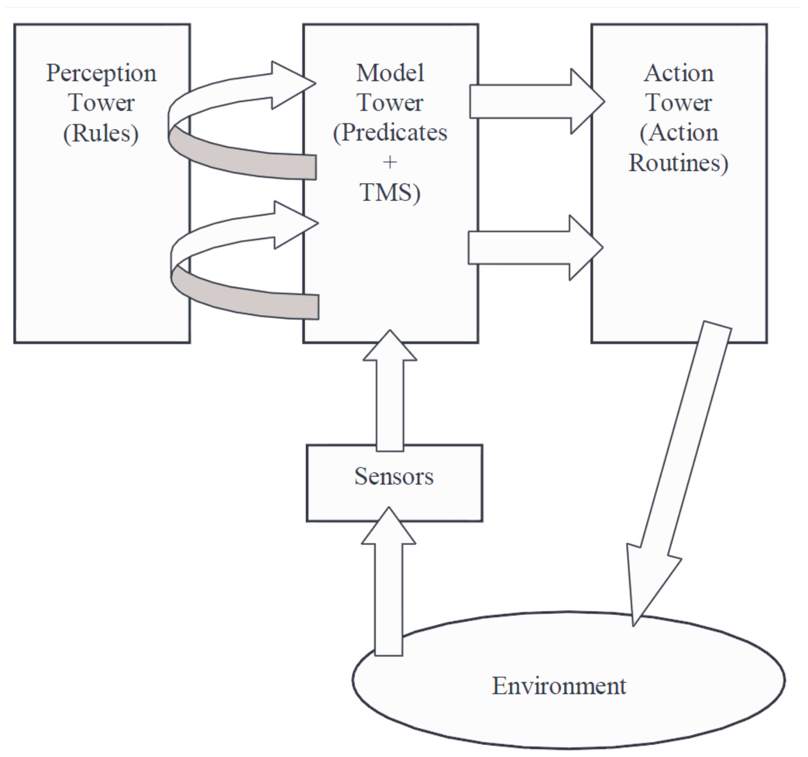

Perception Tower (Rules)
Model Tower (Predicates + TMS)
Action Tower (Action Routines)
Sensors
Environment

Figure 2.1: Triple-Tower Architecture (from [3])

generate new goals in response to the changing current situation and allows multiple actions to be run in parallel in pursuit of several simultaneous goals, thus producing more convincing and believable game agents. The work supports the idea that the T-R approach not only allows programs to be written more naturally, but also produces more natural behaviour from agents.

Katz [90] describes an extension to the T-R paradigm, whereby the boolean logic representing the conditions are replaced by Zadehan (Fuzzy) Logic. The author claims that the mechanism of blocking all previous actions when a higher precedence condition becomes true may not be consistent with the intent of the paradigm. The author demonstrates these claims with Nilsson's own examples and in particular the "bar-grabbing" one from Nilsson's original T-R paper [2]. Here, several assumptions are made. In particular, that the robots movements and rotations will always be exact. This is not always true as, for example, the robot may over or under-shoot its intended position.

Using the proposed changes, the author is able to achieve smoother results which are less reliant on the quality and exactness of the actions involved. These ideas do make sense since the paradigm is intended to provide a continuous goal progression which are provided by the semantics of the paradigm but the use of boolean conditions forces the programs to revert to a digital execution of analogue logic. The results from the author's modifications are encouraging but the simplicity of the approach may be lost in the proposed changes.

Based on Nilsson's triple-tower model [3], the architecture in Coffey and Clark [91] is a hybrid of Beliefs, Desires and Intentions (BDI) and T-R programs. Using BDI affords the authors the possibility to abstract out the rules. For example, we do not have a yes or no condition but a belief which can easily be falsified or an intention to perform an action rather than than an action itself. The authors merged BDI with T-R programs because, after gaining a new desire and intention, the previous intention can be resumed without the need to re-initialize. T-R programs can begin execution at any point, relying instead on the current conditions from the environment rather than retrieving a stored state.

Coupled with this main model, the authors implement communication ability between agents. Communication gives the agents the possibility to share perceptions, desires, and intentions to better coordinate activities. Multiple agents can improve efficiency this way or coordinate with each other to achieve a common goal.

Evidence of the robustness and effectiveness of T-R programming is demonstrated in Mcgann et al. [92] where a T-R based software architecture is used for automated control of an underwater vehicle. The advantage of employing the T-R style here, is that the vehicle is expected to operate in a harsh and unpredictable environment. T-R helps the vehicle avoid and recover from unpredictable events. Using sensors and on board timings, the vehicle is expected to retrieve samples at depths and in environments that humans cannot operate and return to the surface at a specified time.

Gubisch et al. [93] isolate a potential problem where the time taken for one T-R action to complete could be too great, causing one action to possibly cancel the effect of another. An example given is one of a soccer robot where the robot must get in position to kick the ball, then execute the action to draw back the leg and swing towards the ball. The time taken for this action to complete could mean the ball has moved in this time. Here the authors attempt to resolve the issue by extending T-R programming to allow for concurrent execution or multiple actions. This way the leg swinging action can begin before the ball is lined up. When the kick is made the ball should be aligned correctly. Another possible fix would be to change the leg swinging mechanism so that less time is taken for the action to complete or provisions for minor adjustments are made to the mechanism according to current perception of the ball.

There is a lot of potential in T-R programs for adding learning and AI abilities. In this section we highlight some of the different approaches various authors have taken in this task. In autonomics the ability for programs to learn and evolve new abilities has obvious benefits for reducing human maintenance costs and increase self-managability.

Vargas and Morales [94] demonstrate a robot learning new T-R programs by taught actions from humans. Basic actions are first taught to the robot who forms a set of grammars representing the skills. Once grammars are learnt they can be repeated and used to replace repeated sequences of primitive actions. More complex sequences and grammars can then be formed. For example, primitive grammars might be simply rotating to avoid an obstacle or moving forward when the path is clear. More complex grammars might involve moving to a goal location, which could utilize the smaller grammars in part.

These new programs are aimed mainly at navigation oriented tasks. To this end the low-level sensor readings are first transformed into higher level output so that they can be interpreted with physical objects represented as landmarks. Another learning algorithm is presented by Kochenderfer [95] where genetic programming techniques are used to evolve T-R programs and produce programs for solving some problems.

Ramírez [96] uses neural networks to capture new environmental experiences which can be integrated into a learning architecture. Whereas Choi and Langley [97] use a representation formalism is developed called 'teleoreactive logic problems' which support learning. In this method, two knowledge bases exist containing a list of 'percepts' about the environment and a knowledge base containing known actions. They also describe an interpreter that utilizes the logic problems to achieve goals.

2.4.1 Production Systems

A production system imitates some basic logical processes and therefore can provide some form of artificial intelligence. They consist of a list of rules with a sensory condition to check and an action to execute when its associated condition is true.

C Language Integrated Production System (CLIPS) [98] is a production system which is used for building expert systems. It deals with 'facts' and 'rules' where 'facts' are knowledge about the system and form the left hand side (condition) of a rule. When the facts are known then the rule is triggered. In an expert system the rules could be various diagnosed conclusions which are communicated to the user when certain facts are known. A classic example of an expert system would be that of troubleshooting car faults. For example, in CLIPS a rule could be set-up where the user is informed to check the battery or cables if the following facts are true: "The engine will not start" and "The headlights will not come on". Multiple rules like this could form an equivalent of the human expert. JESS [99] is a Java based version of CLIPS.

Originally designed for natural language processing, Prolog [100] can and has been used in other areas such as expert systems and advanced control systems. The program logic is also represented as facts and rules and is initiated with a query. For example, a list of facts about a family history and relationships between each could be inserted. The rules might describe the relationship between two family members, directly or indirectly and a query about whether Frank is a sibling of John could be run. The program might answer yes or no depending on the program rules and known facts.

2.4.2 Suitability of T-R in Autonomics

The T-R paradigm is capable of operating in a non-deterministic fashion. Although ideally each program run should end with the goal, we cannot predict the path it takes to get there and therefore each run could take a different route each time. This is similar in method to a high-level, analogue thought process in a brain. For example a simple task for a human such as loading a dishwasher or opening a door contains several smaller high-level processes such as positioning your body, viewing the object that you want to interact with and extending and coordinating limbs. The human might be interrupted during the process where an unexpected event occurs (maybe a dish is dropped and broken and must be cleared up before continuing) and so the process might be re-arranged to deal with the event before continuing with the original task. The important point is that the original task did not fail just because an unexpected event occurred and no external influence is needed in order to continue. The lack of maintenance required makes the brain 'autonomic' at this high level and is very similar to T-R programming in this way. This ability for T-R programs to manage unexpected events gives T-R programs the impression of self-healing [101].

When we talk about these high-level goals and actions we are ignoring the low-level actions required. For example, coordinating limbs is a very high-level action when the low-level process for this is extremely complex. These low-level actions are handled autonomously however. i.e without the need for conscious thought. A T-R program too contains an ordered list of rules when the low-level actions are 'hidden' when viewing the program at a high level.

2.4.3 Conclusions

T-R programs are very robust and exhibit a self-managing behaviour. This is evidenced when a T-R program meets an unexpected event such as an unknown obstacle in the case of a robot or a sudden disconnection from a network in the case of a higher level application. A well designed T-R program will deal with the event appropriately without ever needing to process code to deal with this specific event; all the time proceeding towards the top level goal.

The production rule list in T-R programs allows focus on the conditions and goals and not the actions, for example, do we have a connection to the network? has a boolean true or false answer. If the connection is lost for any reason then the program should fall back to an earlier state and continue to work on re-establishing the connection. This rule list is very comprehensible both to view and to write and this is one of its major strengths over other production systems, however there is a challenge in the actual composition of a T-R program as detailed in Hawthorne and Anthony [102, 103] and therefore the challenge is primarily one of validation (are we building the right product) and to a lesser degree, one of verification (are we building the product right).

CHAPTER 3

TELEO-REACTIVE PROGRAMMING IN AUTONOMIC COMPUTING

This chapter describes the problem that this research addresses and also the methodology to be used. We also describe the integral part of this, the JTRAF framework, as well as the differences and improvements made, what they do, why they are needed and how to use them.

3.1 Rationale and Method

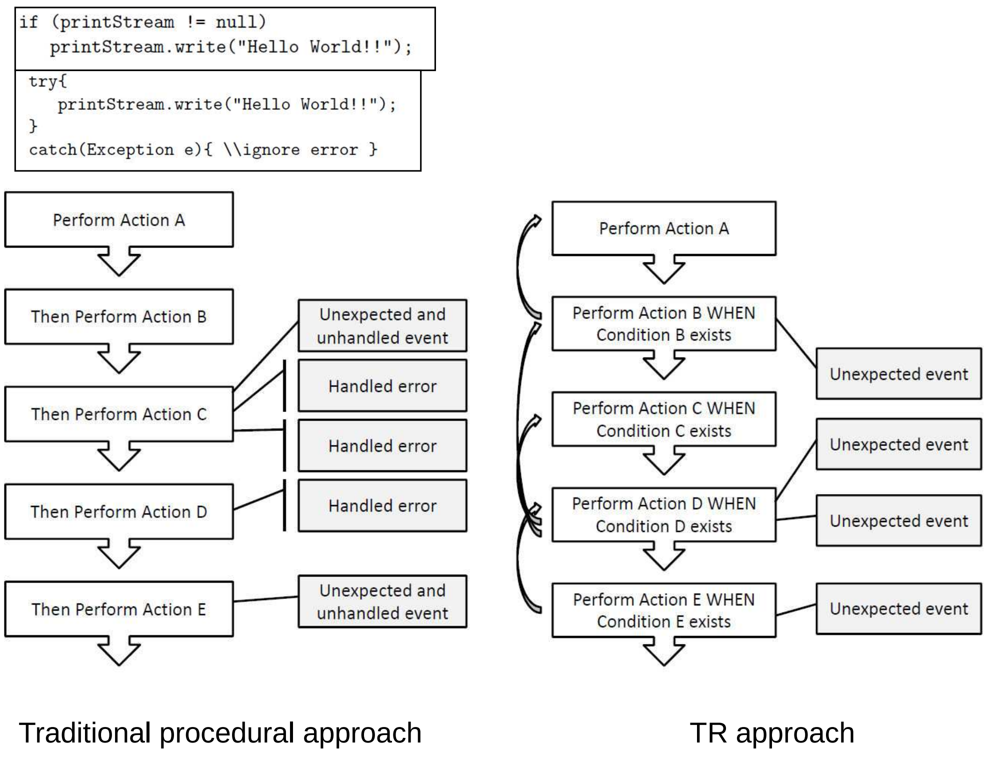
T-R programming seems like an excellent way of approaching the autonomic problem because it firstly provides a natural way to manage the unexpected events which are present on every software system and therefore reduces the maintenance cost considerably. Secondly, the robust basis provides a solid foundation on which to build further self-properties. For example self-configuration in T-R could be achieved in a reliable and natural way without the complexity of a large system. The system designer could add some rules to firstly monitor for changes, then plan a configuration for the change, then execute the plan, the system would then 'fall' back to monitoring again when the new system is complete and stable. This new system could be as simple or complex as the designer desires.

The difference between applying T-R to autonomic systems as opposed to autonomous ones lies mainly in the type of problem to be addressed and the reasons for automation. If an autonomous system typically controls the analogue movement and behaviour of low-level agents with the aim of controlling a system independently of a human, an autonomic system typically controls the configuration and re-configuration and behaviour of high-level management components with the aim of reducing maintenance complexity from the developer point of view.

An autonomous system might control the movement of a robot for example and for this type of application T-R is well suited as a movement action can be interrupted at any point without disastrous consequences. Whereas a high-level autonomic application performs some tasks which cannot be halted (atomic). For example an autonomic system could be updating a record in a database (where database updates would be a high-level activity as opposed to a low-level one), this operation cannot be interrupted as it would cause corruption of data. Therefore, changes in the T-R system may need implementing for this type of situation and for T-R as an autonomic technique in general.

One analogy to describe how T-R programs work would be a satellite navigation system such as ones found in cars or for pedestrians. When a route to a destination is entered into the device, any available satellites are used to discover the current position, then a route to the destination can be planned. If the user deviates from the planned route for any reason a new route is planned from the current position. This means that the current position is constantly checked and that the route is expected to change often. This is analogous to T-R systems where unplanned events are expected and deviations from the original goal are made based on the current environmental context. This non linear progression towards the goal is natural to T-R programs and is quite different to the linear steps taken by most programming languages.

Figure 3.1 shows the contrast between a set procedural language and a T-R system. Often in a procedural language errors and events that are expected to occur are caught with the use of a try-catch style exception handling mechanism. The event is either dealt with at the point it occurred or simply ignored. In all but the most trivial programs, there are many more unexpected events than expected ones. In a procedural approach, these unexpected events will likely cause the program to halt execution entirely as the next action in the sequence can no longer be reached. In contrast the same unexpected event in a T-R system will often cause the program merely to change state, i.e roll back or forward as indicated by the arrows to the left of the T-R program in figure 3.1. All events are handled in the same way (expected and unexpected) so the developer need not write exception handlers for expected events unless specific action is desired.



```
if (printStream != null)
   printStream.write("Hello World!!");
```

```
try{
   printStream.write("Hello World!!");
}
catch(Exception e){ \\ignore error }
```

Perform Action A
Then Perform Action B
Then Perform Action C
Then Perform Action D
Then Perform Action E
Unexpected and unhandled event
Handled error
Handled error
Handled error
Unexpected and unhandled event
Perform Action A
Perform Action B WHEN Condition B exists
Perform Action C WHEN Condition C exists
Perform Action D WHEN Condition D exists
Perform Action E WHEN Condition E exists
Unexpected event
Unexpected event
Unexpected event
Unexpected event
Traditional procedural approach
TR approach

Figure 3.1: Contrast between goal progression of a typical programming language and a T-R program

The mechanism of T-R systems behave similarly to how an analogue human brain processes a goal. A simple analogy is thus used to illustrate this point. A person wishes to load a dishwasher with dishes and utensils etc. so that they will be washed and dried, ready for next use. They begin by placing them in the correct drawers and compartments. Effectively, the person has created an ultimate goal for this task of loading all the dishes and starting the machine. At some point in this process a cup is dropped on the floor and it breaks. The procedure is broken and the human cannot continue until the hazard is removed. A new goal is created to remove the problem before returning to the original goal of loading the dishwasher. Of course this is a very high level view of the process, and the mechanisms (coordinating limbs and muscles whilst observing the environment etc.) are largely hidden but from a high level perspective this process is a lot like a T-R program. In T-R there is a possible set path from an initial state to the main goal but this 'ideal' path is never likely to occur. Instead the currently executing action depends entirely on the current context, creating sub-goals within the main procedure and then returning to other goals where needed. An interruption does not cause a problem like it would if a set path were followed.

In this project and in order to answer our research questions we have developed two use cases using our T-R framework. The two use cases are a simple file sending program which is injected with several faults in order to observe and document the T-R mechanism for processing these faults. We also show how T-R can be used to manage complex datacentres and how the extensions built into the framework can aid in both the development process and improving and extending the existing manager. Shown is the development and run-time process for each.

Initially the intention is to show how T-R can be used as an autonomics technique which these use cases show. We then proceed to demonstrate the extensions to the T-R methodology which were developed and integrated into JTRAF, and how these extensions expose problems and lessen the effort of developing and maintaining a T-R program.

3.2 The Java Teleo-Reactive Autonomic Framework (JTRAF)

The Java Teleo-Reactive Autonomic Framework (JTRAF) is a Java framework which is designed to make it simple for developers to use the Teleo-Reactive paradigm in high level and autonomic applications. The framework allows rules to be easily written and interpreted, thus minimising design effort.

JTRAF can be used to develop a complete application as a T-R realisation or the T-R method can be used in a sub-program where T-R is suited to a sub-goal and not the main application. It could be used in a legacy project where a suitable solution had not previously be found.

From the point of view of this project, JTRAF is an easy and convenient way of developing the T-R simulations which are used to answer the research questions, i.e. the JTRAF library is developed separately to the main application and is referenced and used in the main project and any other project unchanged. This is convenient because a T-R program cannot directly be written in a high-level language such as C or Java but instead requires an interpreter which JTRAF provides. If a change to JTRAF is needed then it should be made without having a negative effect on any application that uses it. To help ensure this, the interfaces of JTRAF remain unchanged throughout development.

Java is used in development of JTRAF because it is a high-level language, so this is applicable to the high-level applications which we aim to demonstrate the use of T-R in and could be incorporated into real-world projects and applications in development. Java is an object-oriented language which is convenient for modern applications which use JTRAF. It is also convenient for development as the different aspects of the framework can be represented as objects and classes which can then be extended to allow for application specific methods whilst confining the framework specific code to a higher level parent class.

A T-R program is essentially a prioritised list of production rules, and a rule

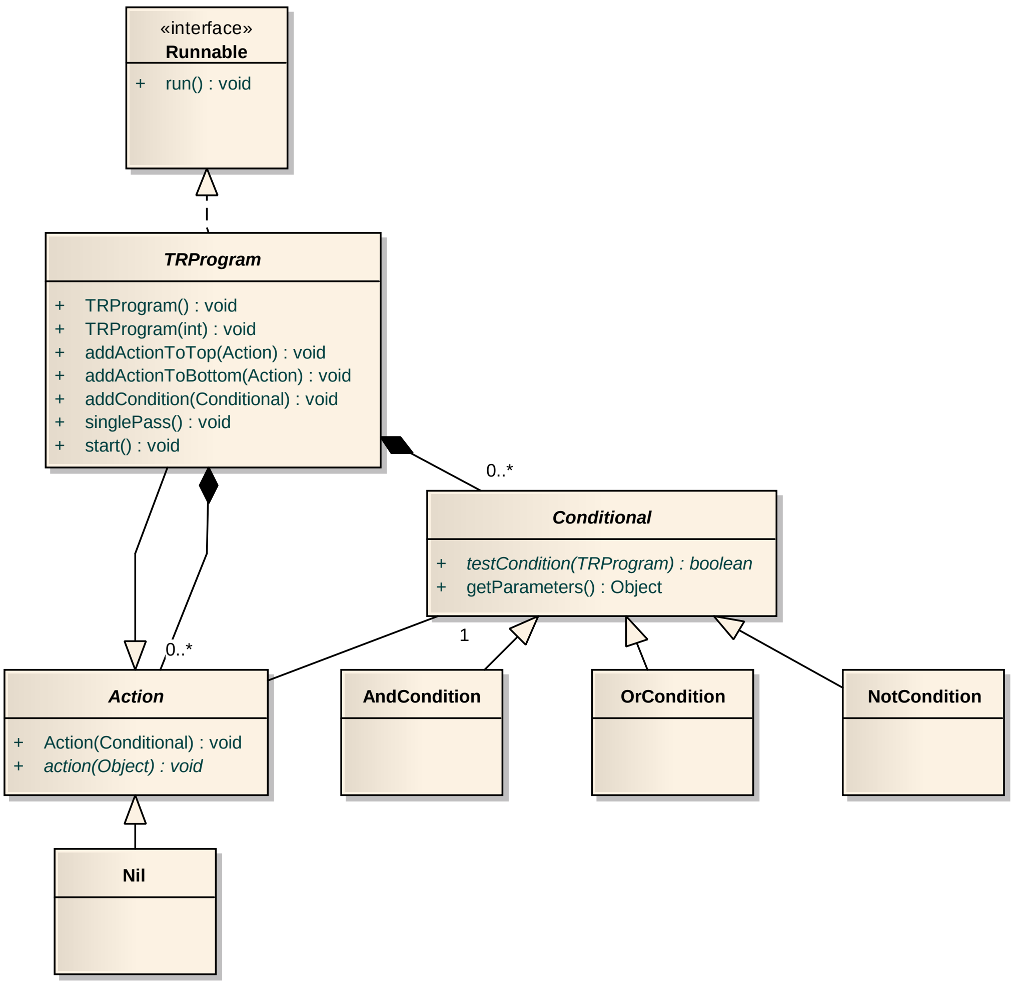


Figure 3.2: Teleo-Reactive framework (JTRAF) class diagram

consists of a true or false condition followed by an associated action. JTRAF therefore provides an easy way to extend conditions and actions and associate the two (forming a rule) and order the rules so that they are given priority. Once the T-R program is complete it can be run as a background thread or as the main program.

3.2.1 Creating a Rule

A rule is created by first constructing a condition and then constructing an action with the newly created condition object as an argument. Both Action's and Condition's are abstract types and must be extended to create application specific objects. This ensures that the elements which are required by JTRAF are constrained to the Action and Condition super classes. The developer is freed from the operational concern of the framework and can concentrate on specific rule construction.

As can be seen from figure 3.2 an Action only ever accepts a single Condition as an argument, however there are three built-in extensions to the Condition class. These are, AND / OR / NOT conditions, which logically combine, logically OR two conditions and negate a condition respectively. These three conditions can be used to create some complex nested logic if need be i.e. the result is a single condition which may or may not have several levels of complexity.

Throughout the development of JTRAF a Test Driven Development (TDD) methodology was employed. One of these tests simply adds to a central counter until the counter reaches 5000. Adding a value to a counter here was a useful test for Validation and Verification of a T-R JTRAF program where the test is designed to show how a T-R program can be composed in JTRAF. The intended, simple T-R program is shown in figure 3.3, and the simple steps used to construct this program are shown here:

```
Condition isTrue = new IsTrue();
Action add10 = new AddAction(isTrue, 10);
Condition is100 = new NumCondition(100);
Action add100 = new AddAction(is100, 100);
Condition is1000 = new NumCondition(1000);
Action add500 = new AddAction(is1000, 500);
Condition is5000 = new NumCondition(5000);
Action finished = new Nil(is5000);
```

There are just two custom types here, AddAction, which accepts two parameters, one condition which is passed to the parent class (Action) and an amount to add to the central counter. NumCondition returns true if the counter is greater than the value specified and false otherwise. Nil is a built-in Action and does nothing. It is often used at the end of a T-R program, i.e. when the goal condition is true. IsTrue is a built-in condition which returns true always and is often used to initiate a program, i.e. start the first Action when no other Action is possible.

AddAction and NumCondition classes begin like this:

```
public class AddAction extends Action
{
    private int _incAmmount;

    public AddAction(Conditional conditional, int incAmount)
    {
        super(conditional);
        _incAmmount = incAmount;
    }
...
```

and...

```
public class NumCondition extends Conditional
{
    private int _ammount;
    private TRProgram _trProgram;

    public NumCondition(int ammount)
    {
         _ammount = ammount;
    }
...
```

respectively.

There is no Rule type because each Action is associated with a Condition and so this construct effectively forms the rule.

3.2.2 Creating a T-R Program

$$
\begin{aligned}
Is5000 &\longrightarrow Nil \\
Is1000 &\longrightarrow Add500 \\
Is100 &\longrightarrow Add100 \\
T &\longrightarrow Add10
\end{aligned}
$$

Figure 3.3: Simple JTRAF composition test (Counts to 5000)

The rules have been created but the program in figure 3.3 still needs to be ordered and composed. First we create a TRProgram instance and add the rules to it, like so:

```
TRProgram testProg = new TestProg(5);


testProg.addActionToBottom(finished);
testProg.addActionToBottom(add500);
testProg.addActionToBottom(add100);
testProg.addActionToBottom(add10);
```

where testProg is an instance of TRProgram.

Figure 3.2 shows two methods to add a rule 'addActionToTop(Action)' and 'addActionToBottom(Action)'. These methods exist to avoid confusion as to where the rule would be placed within the T-R structure, i.e. a method 'add(Action)' would not be informative to the developer.

The condition that the action contains is automatically added to the condition list within the TRProgram instance and can be tested individually from a unit test for example as can individual actions. This is a useful feature for test driven developers since the components can be tested in isolation rather than waiting for the entire T-R program to be composed and executed before finding any errors. See [104, 105] for more about Test Driven Development (TDD).

Figure 3.2 also shows that TRProgram implements the Runnable interface meaning it can be run as a background thread. This is useful if only a small part of the application would benefit from the advantages of T-R. A delay time (milliseconds) can be set through one of the constructors to TRProgram and the thread is executed by calling start().

The singlePass() method performs a single evaluation run through the T-R program which can be useful for executing as part of an external / custom thread or part of a debugging or testing strategy for example.

3.2.3 Extensibility through sub-T-R programs

TRProgram extends the Action class itself. This means that the actions which are triggered need not be simple actions but could be entire T-R programs themselves. This is similar to the Composite design pattern of Gamma et al. [106] and promotes extensibility whilst keeping individual T-R programs short and manageable.

3.2.4 Temporal analysis extension

The temporal analysis panel (TAP) allows the developer to view the state of any condition in a T-R program over time. Each condition is represented as a progress bar showing what percentage the condition has been true over the course of the program run. The bars also show the logically joined conditions, so that the percentage true time for any condition is shown and also the percentage true time in conjunction with the other condition logically joined with it. The layout of the T-R program and current state of the conditions is also shown in real-time. Shown in figure 3.4 true conditions are in green and false are red, the currently executing action is underlined.

When developing the TAP tool, we observed that the visual information from the condition indicators showed decreasing levels of accuracy as time increased. This is because the proportion of time a condition has been true varies depending on the time scale used. For example a condition true for two minutes of a four minute period is a 50% proportion whereas two minutes of two hours is $\approx 1.6\%$. This meant that if a T-R program had been operating for a long period of time and a condition state changed, it may be impossible to observe from the progress bars that any change occurred.

To aid in this problem, exponential smoothing was used to apply exponentially decreasing weights on values over time. This provides greater visually sensitive feedback to the developer and it also means that an exhaustive history of previous events is not needed to achieve this since all the previously smoothed data is held in the most recent smoothed sample.

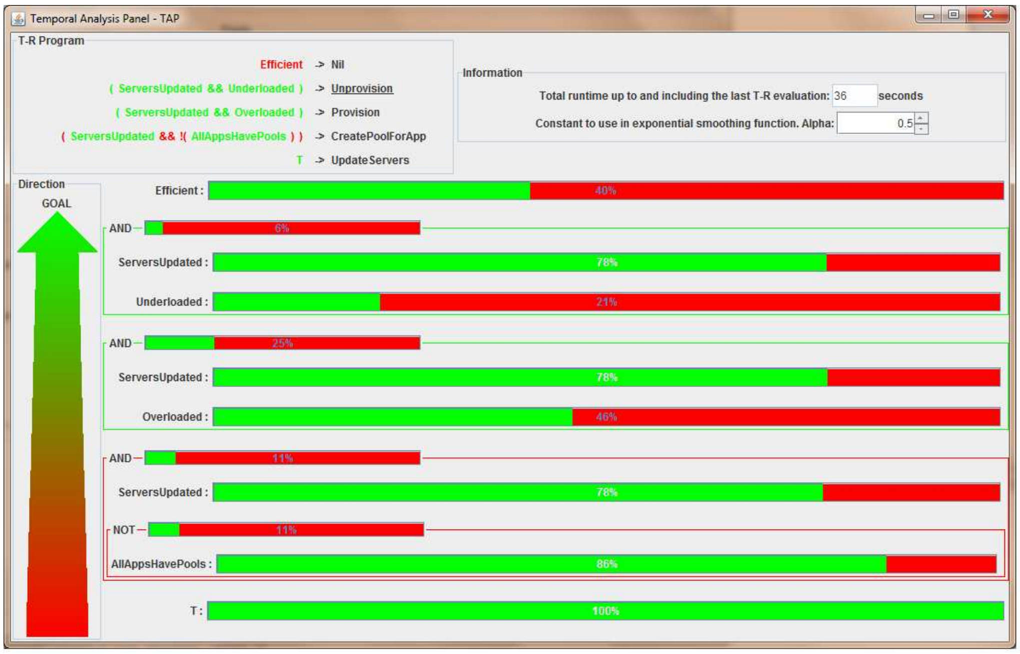
TAP is part of JTRAF and is automatically generated as per the T-R program which it is called from. So the TAP screen of figure 3.4 was not developed for the datacentre use-case specifically but generated by JTRAF automatically where any T-R program can ‘generate a TAP’ with a simple call to setVisible():

```
mainTRProgram.getTAP().setVisible(true);
```

or whatever the TRProgram instance happens to be called.

The same data that is shown graphically in TAP is also automatically logged to an excel file. The developer can then view the logged events with a global view of the program run or to generate a different view of the data from a graph or table.

The next section explains that exception handlers are used to catch run-time errors and help the T-R program continue to operate as it should. However it is possible and quite likely in fact that the same run-time error in a particular action will re-occur indefinitely until it is physically corrected. This would mean the program never proceeds past the rule which contains the error and thus never reaches it goal. Another example of these logical errors is where a rule is skipped entirely because the previous action enabled a condition which in turn enabled a rule which has higher precedence than the skipped rule. This may not be the program designers intentions and the skipping of a rule might prevent the goal from being



```
                                    Efficient  -> Nil
              ( ServersUpdated && Underloaded )  -> Unprovision
               ( ServersUpdated && Overloaded )  -> Provision
    ( ServersUpdated && !( AllAppsHavePools ) )  -> CreatePoolForApp
                                            T  -> UpdateServers
```

Total runtime up to and including the last T-R evaluation: 36 seconds

Constant to use in exponential smoothing function. Alpha: 0.5

Figure 3.4: Temperal analysis screen (generated from the datacentre use-case)

reached again.

What is important to recognise here is that although run-time errors are reduced with T-R, logical and validatory errors can actually increase and it can be much more difficult to find the source of these problems. These problems and possible methods of prevention are described in Hawthorne and Anthony [103, 102].

TAP was designed to help alleviate this problem by providing greater feedback to the developer. If for example a condition is always false over a one hour program execution and the developer knows the condition should be true for part of the time, then the problem can be immediately highlighted, assessed and fixed. The problem is likely to be contained in the preceding action which would be difficult to locate without TAP.

3.2.5 Differences between low-level T-R and JTRAF

One advantage of creating a high-level interpreter for T-R is the availability of exception handling constructs. In JTRAF all actions and conditions run within exception handling constructs and *any* expected or unexpected error is caught at run-time. When a run-time error has occurred and is caught, the action simply ceases to execute. The evaluation of the conditions then begins again. In this way, an error introduced by the application developer will not cause the entire T-R program to end.

If, after the error occurs and the conditions are re-evaluated, the same error occurs indefinitely then the program would enter into an infinite loop and never reach its goal. The T-R developer would then consult the TAP and be able to observe that all conditions after the error have never been true. The developer would then know the T-R program is faulty and using TAP they would know where the fault is likely to reside, i.e. the action or condition preceding the indefinitely false conditions. The developer need not write their own exception handler as JTRAF will catch exceptions automatically.

It could be argued that a similar exception handling effect could be achieved by containing an entire program, written in another language in an exception handler but the program would then need to restart in its entirety no matter how big or small the occurring error is. By using T-R which adds handlers around individual actions and conditions, the program only needs to revert to a an earlier state and rule in the event of an error and does not necessarily need to restart in its entirety. If the opportunity arises the state can alter to a rule with higher precedence as well.

Low-level T-R programs are designed to execute an action continually until it ceased to hold the highest priority, true condition. If this condition becomes false then the action would immediately stop and a lower action (which holds the highest true condition) would take its place. This means that the conditions are continually evaluated separate from the currently executing action i.e. the T-R program and its conditions are evaluated in a separate thread of execution to the running of an action and any action can be interrupted at any moment during its execution. The alternative, and the method used in JTRAF, is to evaluate conditions from the top down and execute the action with the highest priority as with traditional T-R programs. However, actions cannot be interrupted and must reach the end of execution before the evaluation begins from the top down again. i.e. the evaluation and running of actions all happen in a single thread of execution.

If an action requires more than one iteration to fulfil its goal, the developer need only write the code for a single iteration for the action. The same action would be executed as many times as necessary by the T-R program, i.e. for as long as the action is associated with the highest true condition. Effectively, the T-R program creates an inherent loop construct. This does have a major drawback however in that an action that contains some error and executes indefinitely would effectively lock the program as no evaluation could take place until the faulty action ends. A traditional T-R program does not suffer from this problem as conditions are always evaluated regardless; but also has a major drawback in that the continual evaluation will impact negatively on processing time, which could be a problem in high-level and processor intensive applications.

A low-level T-R robot for example might be required to apply constant current to motors to keep them turning and keep the robot moving forward, therefore any pause for evaluation of conditions would likely cause slow or jerky movement or otherwise prevent the motors operating correctly. It is therefore vital that there is no evaluation period in the T-R program execution. In all the T-R programs we have developed through JTRAF however, the type of processes run in these high-level programs have never warranted the ability to interrupt an action. This has never been an issue and in fact it can be dangerous to interrupt an action before completion. If the action was updating a database for example then an interruption could lead to an incomplete file transfer or a corruption of the database.

In every previous high-level T-R program written during this project there has never been a need for the more traditional continual evaluation of conditions. However, JTRAF allows for a future project where it may be necessary and the framework provides the option to switch to a more traditional method where the condition evaluation and action execution are run as two separate threads, thus allowing actions to be interrupted at any point.

The research questions are now revisited in more detail and related to the research methodology.

3.3 To what extent can Teleo-Reactive programming techniques (T-R) be used to construct autonomic management systems and what are the benefits of doing so?

This research question requires that the effectiveness of T-R as autonomic control logic is evaluated in a number of differently characterised application scenarios. It is necessary to identify which types of application, and which specific types of control scenario can benefit from the T-R approach, and also to identify the boundaries and limitations of this approach.

We believe that the T-R system has a lot of potential for use in autonomics because we argue the T-R idea parallels the autonomic nervous system very closely. A human solves a problem by addressing high level goals such as 'open door', or 'put on shoes' without concern for background tasks and low-level mechanisms such as breathing and coordinating muscle movement etc. Our only concern is solving the immediate problem and not the exact mechanism for doing so. These mechanisms are largely handled automatically so that our conscious thought can be aimed at high-level goals.

The T-R system works in a remarkably similar way where goals and conditions are the focus and not the low-level mechanics. Conditions and goals are the driving force of T-R as they inform the system as to which action to execute and which state to be in. Actions attempt to satisfy higher priority conditions but the system does not need to know how or even whether the action is successful or not. It only needs to know which conditions are true and which are false in order to progress towards its goal. Essentially, the low-level actions in T-R are parallel to autonomic actions in a nervous system because, whilst sometimes essential, they are only indirectly involved in goal completion. The conditions are the true indicators of the state of progression towards a goal. And so, as in the human system, we consciously focus on the goals and conditions and not the actions.

3.3.1 Approach

The file sending application in section 5.1 is a high-level system which uses the T-R approach. We can therefore use this demonstration to show the extent of T-R capabilities in high-level applications and to validate and justify its use.

We can use this demonstration to answer the question in a broken down form, i.e. we can split autonomics into its four main self-properties and question whether T-R produces self-healing behaviour, produces self-configuring behaviour etc. We will test the self-healing behaviour of T-R by injecting faults into the running system and observing the behaviour.

If designed well it should be able to recover from expected events and may even recover from some unexpected ones, but this is difficult to demonstrate since we cannot prove that the fault injected was not planned for. However we should be able to provide a good indication of the healing process both in this particular demonstration and in other T-R / JTRAF examples.

We will also demonstrate the self-optimisation behaviour by increasing and decreasing the noise levels in communication and observing the effects. The system has been designed to adjust the packet size according to the successful receipt of previous packets. We also comment on the applicability of T-R in autonomics as a whole, describing its benefits and ease of use.

3.4 What are the difficulties in T-R programming and how can they be reduced?

The benefits of T-R as an approach to building software is clear, but it is also clear that this approach is not beneficial in every situation. It is also clear that whilst some problems can be reduced and the system is generally more robust, some problems are more prevalent as shown in Hawthorne and Anthony [102]. This project shows how these concerns and problems can be addressed.

3.4.1 Approach

A concern with this technique is that a T-R program might not be very extendible and is only really applicable to small problems. This concern however is not true and although keeping programs small and concise is a good idea for readability and maintainability we can expand programs to any level with sub-T-R programs. Section 4.2.4 demonstrates how a T-R program can be extended with sub-T-R programs without affecting the structure of the main T-R program. We also show how TAP can be an integral part of the development process, both to inform the developer visually of the current structure and state, and as a visual aid to the compositional structure of the T-R program over a time period.

3.5 What improvements can be made to extend the benefits or increase the ease of adopting T-R programs?

This question is essentially saying that there are several benefits and several problems associated with T-R programs. Do the benefits outweigh the problems and so encourage the use of this method for developers? If the benefits are not sufficient, what can and has been done to reduce these problems?

3.5.1 Approach

In section 4.2.4 we outline the problems in composing programs and show how a program is typically written using the TAP mechanism. TAP improves the ease of composing T-R programs significantly.

In section 5.2 there are three T-R managers which each perform the same task of running the datacentre simulation. The first manager is the basic program, it is very simple in that it contains only a few rules and only one main T-R program. It does however do all that is necessary to the complete the goal.

The second program extends the basic manager with a sub-T-R program which performs extra decisions over which pool should receive a server upon multiple pools being overloaded. It will also remove a server from a pool and apply it to another pool if its requirement is higher and if there are no idle servers available.

The third program alters the second, searching for trends in the number and size of requests and applies and removes servers to pools based on this trend.

Each of the managers will be used to run the datacentre using an identical script so as to provide the same input data each time. TAP will automatically log the state of each condition in the manager and the data can then be compared in order to assess the changes. For example one of the managers might show that the 'efficient' condition is true for a longer period of time overall over the program run than the other managers whereas another manger is an improvement in certain situations but not others. These results will be documented and compared thus showing how TAP can aid improvement to existing T-R managers.

In addition to the benefits which TAP provides, we also have plans to build advanced composition techniques into JTRAF as described in Hawthorne et al. [107]. Here T-R programs compose themselves based on knowledge of available actions and past event knowledge. This will lead to less error prone programs and also increase autonomicity since the programs will manage themselves to a large degree. This is a huge advantage of JTRAF since this and any other changes can be made directly to the framework which can then be taken advantage of by all JTRAF developers.

CHAPTER 4

JTRAF USE-CASES

4.1 Teleo-Reactive File Sharing Application

The example presented here is a simple simulation of a file sending application presented from the point of view of the server side. The example extends JTRAF to simulate a simple protocol which we assume has little or no error handling abilities. The program contains several rules pairs leading to its main goal. For the sake of simplicity the file is always the same and is of a fixed length, with the aim being to send the file to the currently connected client. The example demonstrates how unexpected events such as random client disconnection and noisy communications are recovered from. This example is not designed as an accurate representation of networking components, it only serves to highlight some of the features and benefits of the T-R paradigm.

4.1.1 Design

The file sending application was developed before the TAP tool was created and therefore the T-R program for the application was developed with a trial-and-error method. The program is still correct with respect to completeness but this development would have been easier and quicker with the TAP tool. For example, some of the problems written about in Hawthorne and Anthony [102] such as 'T-R deadlocks' and 'prioritised skipping' were evidenced in the development of this example. Figure 4.1 shows the completed file sending T-R program.

$$\begin{aligned} IsFileComplete &\longrightarrow Nil \\ IsFivePacketsSent &\longrightarrow ChangePacketSize \\ IsConnected &\longrightarrow SendNextPacket \\ IsAccepting \wedge IsWaiting &\longrightarrow Connect \\ T &\longrightarrow Accept \end{aligned}$$

Figure 4.1: T-R program for file sending application

T-R Elements

The goal of the example is the same as the top level condition, to finish sending the file to the connected client. With the file sent the program does nothing until the top level condition turns false again. Figure 4.1 shows an alternative view of the 'Teleo-Reactive Program State' section from the example.

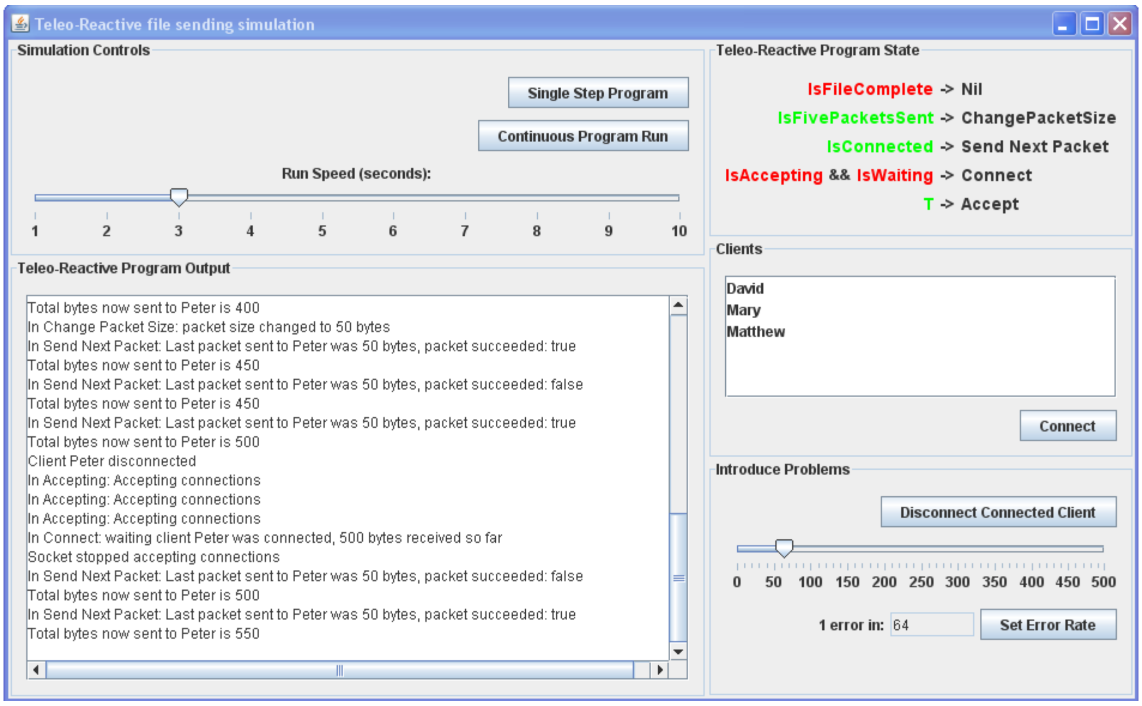
The conditions in the example are:

- **T** - As this is always true its action will always execute if no higher precedence condition is true.
- **IsAccepting** - The example simulates a socket which accepts incoming connection attempts. This condition returns the accepting state of the socket.
- **IsWaiting** - This condition reports whether there is a client in the waiting state.
- **IsConnected** - The client is connected after it is moved from waiting to the connected state.
- **IsFivePacketsSent** - Returns true when a multiple of 5 packets has been sent to the connected client.
- **IsFileComplete** - True once the whole file has been transferred to the connected client.

The actions in the example are:

- **Accept** - Simply turns the accepting state of the socket to on.
- **Connect** - Any waiting client is connected and the Accepting state is switched off.
- **Send Next Packet** - A packet is sent to the connected client at the current packet size.
- **Change Packet Size** - The packet size is adjusted using a simple algorithm based on statistics from the previous packet sending attempt.
- **Nil** - Performs no action.

GUI



```
IsFileComplete -> Nil
IsFivePacketsSent -> ChangePacketSize
IsConnected -> Send Next Packet
IsAccepting && IsWaiting -> Connect
T -> Accept
```

1 error in: 64

```
Total bytes now sent to Peter is 400
In Change Packet Size: packet size changed to 50 bytes
In Send Next Packet: Last packet sent to Peter was 50 bytes, packet succeeded: true
Total bytes now sent to Peter is 450
In Send Next Packet: Last packet sent to Peter was 50 bytes, packet succeeded: false
Total bytes now sent to Peter is 450
In Send Next Packet: Last packet sent to Peter was 50 bytes, packet succeeded: true
Total bytes now sent to Peter is 500
Client Peter disconnected
In Accepting: Accepting connections
In Accepting: Accepting connections
In Accepting: Accepting connections
In Connect: waiting client Peter was connected, 500 bytes received so far
Socket stopped accepting connections
In Send Next Packet: Last packet sent to Peter was 50 bytes, packet succeeded: false
Total bytes now sent to Peter is 500
In Send Next Packet: Last packet sent to Peter was 50 bytes, packet succeeded: true
Total bytes now sent to Peter is 550
```

Figure 4.2: GUI for file sending application

The simulation GUI controls enable the user to set a run speed by changing the run speed slider and clicking the 'Continuous Program Run' button. This sets the simulation to perform a full pass through its action list every X seconds. The 'Single

Step Program' button performs a single pass at each press and turns off automatic running.

The program state area gives visual feedback on the current progress in the T-R program. In the GUI a 'true' condition is coloured green, whilst a 'false' condition is coloured red. The action associated with the highest precedence 'true' condition is executed and will continue to execute at each pass while it remains so.

There are four possible clients that can be connected to the file sending service. Clicking on one of these four names followed by 'Connect' performs the connection attempt.

The 'Introduce Problems' controls are used to inject 'unexpected' conditions. This includes a random disconnection of a connected client and the introduction of degraded communication. The level of degradation is set with the slider and 'Set Error Rate' button.

4.1.2 Working Through the Example

In this test, we make a connection and begin sending the file to this client in blocks. We demonstrate how the program copes with the unexpected events of disconnection and changing quality of service. This simple example is intended to illustrate the benefits and robustness of the approach.

When the example application is started the only 'true' condition is the first 'T' condition. On the first pass, the accept action will be executed, making the 'IsAccepting' condition 'true'. This will be the only executed action on subsequent passes until a higher precedence condition evaluates to 'true'. If we now connect one of the clients, the 'IsWaiting' condition becomes 'true'. The conditions for the action 'Connect' are now completely satisfied and takes precedence over any lower actions. Its execution on the next pass will connect the waiting client, stop the socket from accepting connections, and cause 'IsConnected' to return 'true'. The service will continue to send packets to the client unless a higher precedence condition becomes 'true'.

There are a variety of ways in which the connection to the client can be lost. Clicking 'Disconnect Connected Client' causes the only 'true' condition to be 'T' again. The program will then work back to the point where the problem occurred and continue sending packets until the entire file is sent. The file send is resumed at the point where the last successful packet was sent. If another client connects before the first client, they must wait until this client disconnects.

If, after five packets have been sent, very few packets reached their destination, it might be beneficial to reduce the size of the packet so that more packets are sent successfully. This is because the complete file may never get sent if there is too much noise causing too many errors in the network connection sending each packet. A smaller packet size has a higher chance of being error free, and at least a few of these may succeed in sending. Every multiple of five packet sending attempts the service has the opportunity to adjust the packet size to suit the current communication quality and keep the service running at optimal levels.

By changing the error rate in the simulation we can show how this works. The packet size will increase or decrease as appropriate. Once the 'IsFileComplete' condition is satisfied no other action will be performed and the goal is complete.

It was simple to add self-optimising behaviour to the program, which is demonstrated by the changing packet size program function. We had not intended to implement self-optimization within the first version of the example, but it turned out to be very simple to add this function. Using our framework, constructing the application was remarkably easy, and with future iterations of the framework (adding further robustness and inherent validation checking), this will be an easier task.

A more elaborate example scenario could be chosen with the opportunity to introduce a greater variety of problems and some of further features of the framework such as ease of reuse and recursion. However, the example is simple enough to demonstrate the main advantages which a more complex scenario might have blurred.

4.2 Teleo-Reactive Autonomic Datacentre

The datacentre is commonly used to house many of a companies IT related operations. There is a high requirement for access to services to be reliable, secure and smooth. The number, capacity, and effectiveness of servers associated with a particular application indicates the capability of the datacentre to deal with incoming requests. A noticeable drop in performance would be evident if the number of requests for a service far exceeds its capacity to deal with it. It may also be desirable to increase or decrease the capacity for one application at the expense of another, or to provide enough capacity for an application before it becomes "over/under-subscribed" either by analysing historical data and/or recent trends in request data.

These requirements lead to a desire for the management of a datacentre to be computerised and automated as much as possible in order to reduce human maintenance costs and to operate optimally or as close to optimal as possible with the current resources. An effective autonomic system should be able to effectively manage these requirements and adapt to unexpected events and changes in the environment and in the system itself.

Many proposed autonomic datacentres utilise complex architectures [50, 108, 109], making these techniques and architectures difficult or impossible to replicate for any scenario for which they were originally designed. This section proposes the T-R paradigm as an effective autonomic datacentre manager which is much less limiting. In other words, although we are using a datacentre to demonstrate the capabilities of T-R, it is not limited to this scenario.

The T-R datacentre as a simulation has been designed as three distinct parts. The model, containing the current state of the datacentre and its elements. The T-R manager, which contains the behaviour and decision making automated processing of events. The script driven automated request generator is used to provide a simulation of requests for services within the datacentre.

4.2.1 Model

The datacentre model does not contain any behaviour itself, it is a logical representation of the datacentre state. The model represents a very simple datacentre scenario where the simulation focuses on the running efficiency rather than security, power requirements or other aspects; the model includes elements with this goal in mind only. Although simple, the datacentre is intended to inspire extensions in order to fit more complex scenarios. Figure 4.3 is a UML class diagram representing many important aspects of the model.

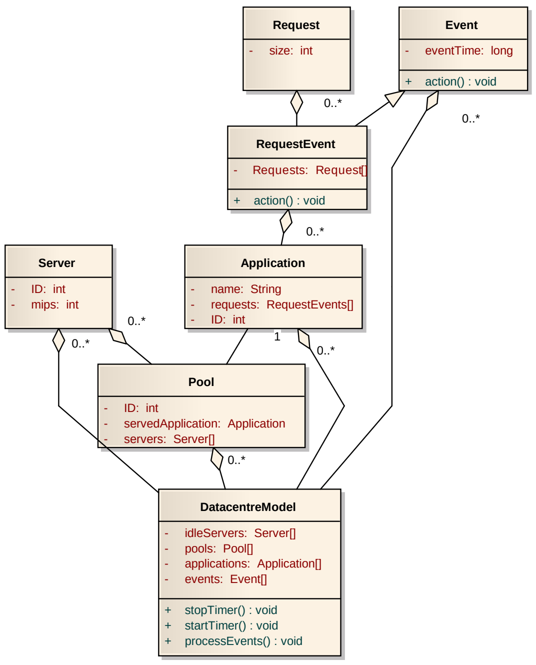


Figure 4.3: Datacentre model

- DatacenteModel - Contains the applications which are present in the model, these are the services which the datacentre is capable of serving but may or may not be associated with a pool yet. It also contains a list of created pools, events and servers. Unlike applications, servers which exist in the datacentre model are here only if not associated with a pool, they cannot be in two places at the same time. Therefore, servers which exist in the model can be considered idle, i.e. not being utilised.

The datacentre model is also responsible for processing any of the listed events it contains which have expired. The event list is checked periodically for events where the eventTime has been proceeded by the current time. If it has the action associated with the event is executed.

- Events - These form the logical connection and interface between an external script and the datacentre, however it is possible to create event objects internally as part of a unit test for example. All events include an event time, which indicates but does guarantee the time, offset from the start of the program run when the event should occur. Each event also includes an action() method which is executed by the datacentre model when the event time expires. Action() should not include any large operations because this is the task for the datacentre manager not the model. For example, request event actions simply add requests to an application, how this is interpreted is down to the manager.

- Requests - An application can contain many requests, representing the current demand. Request size is measured in MIPS (million instructions per second) and are essentially the driving force for the manager. i.e. as the requests increase, more servers may need to be provisioned. So too, as the number of requests decrease, the number of servers available for an application may need to decrease to be made available for another application.

- Application - Represents the requested elements. The data-centre model can contain many applications, these are a list of all the available applications. A pool however can and must contain one application.

- Pool - A logical construct which relates an application with a list of servers. A server can only be added from the list of idle servers so if there are none free, the pool must wait for one to become available or the manager may try to take a server from a less overloaded pool and add it to the idle server list first. This is dependant on the complexity of manager loaded.

- Server - A server is an element which is capable of dealing with incoming requests. A pool links the servers to an application such that, as requests for the application arrive, there must be enough capacity associated with the application in order for seamless communication to occur, i.e. enough servers for the requested application, where server capacity is measured in MIPS. Each server can be in one of three states as shown in figure 4.4, idle, provisioning and In pool. This shows that a server cannot immediately be assigned to a pool, but must first join the provisioning queue for a pre-determined length of time.

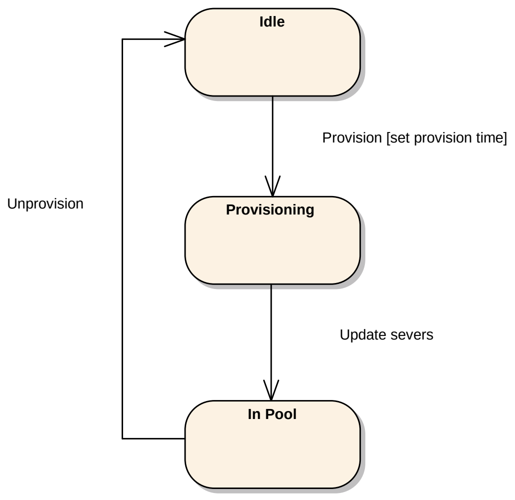


Figure 4.4: Server states

4.2.2 The T-R datacentre manager

There is a necessity for large datacentre managers to operate automatically because the amount and frequency of decision making would be too much for large teams of humans to manage. The manager also needs to be robust enough to manage various states of the system without much or any possibility of failure as this may result in a lengthy downtime of the system. The T-R datacentre manager provides all the logic, decision making and processing in an autonomous way and because the manager is a T-R type paradigm it is very stable and able to recover from unexpected events which may cause problems for other systems performing similar tasks.

The manager needs to be able to delegate servers to pools that need them and remove servers from pools which contain too many. This should be formulated into a true or false condition and set as the main goal. As the rest of the program is developed each rule should work directly or indirectly towards this goal. The goal may never be met if the frequency of request rate change is always higher than the speed the manager delegates servers for example but the goal should be the main target nonetheless. The T-R system looks at how the goal is effected by a recent event, and not on event itself, this way we can manage a much wider range of events as we do not need to know the cause of the condition change, only that some event has caused it. This gives the T-R manager increased robustness and is arguably the best approach since unexpected events can never be planned for anyway. The complete T-R program is shown in figure 4.5 and the process involved in composing the datacentre though the TAP mechanism is shown in section 4.2.4.

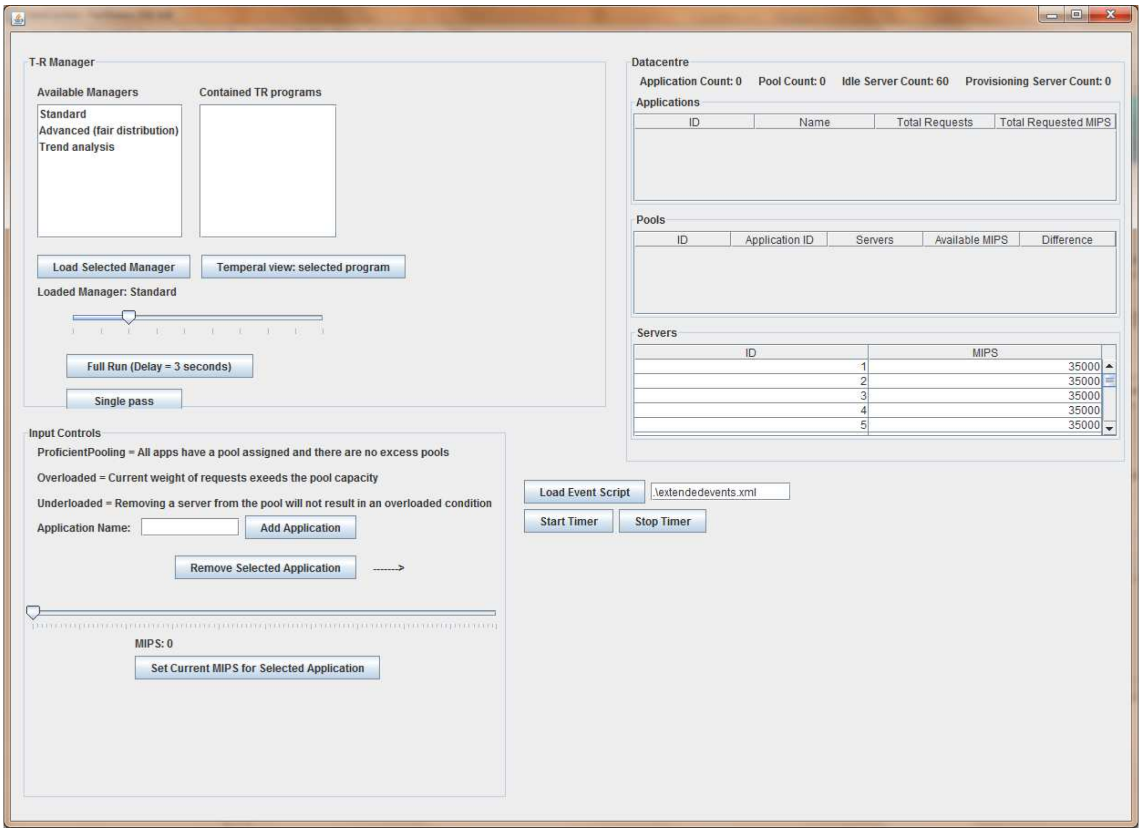
$$\begin{aligned}
Efficient &\longrightarrow Nil \\
ServersUpdated \land Underloaded &\longrightarrow Unprovision \\
ServersUpdated \land Overloaded &\longrightarrow Provision \\
ServersUpdated \land \neg AllAppsHavePools &\longrightarrow CreatePoolForApp \\
T &\longrightarrow UpdateServers
\end{aligned}$$

Figure 4.5: Full datacentre T-R program

4.2.3 Script-driven Automated request generator

Figure 4.6 shows the main GUI for the datacentre. The lower most slider allows simulated request amounts for applications to be manually set and changed. The manager could then respond to the change and the T-R state changes could be observed. This presented a problem, in that the regular changes to the simulated requests is a monotonous and time consuming task. Of course the request amount could be chosen at the start for each application and could remain fixed at this value for the duration of the simulation but this would not be very realistic input for any simulated datacentre scenario. Furthermore, test input would be difficult to repeat over a range of different managers.

A scripting system was implemented to allow the developer to construct an external xml script simulating the requests. Figure 4.6 shows the mechanism to loading these scripts, i.e. the name of the script is chosen and loaded. The script is processed in a separate thread which is started and stopped with the appropriate buttons.



Standard
Advanced (fair distribution)
Trend analysis

Loaded Manager: Standard

ProficientPooling = All apps have a pool assigned and there are no excess pools

Overloaded = Current weight of requests exeeds the pool capacity

Underloaded = Removing a server from the pool will not result in an overloaded condition

Application Name:

MIPS: 0

Application Count: 0 Pool Count: 0 Idle Server Count: 60 Provisioning Server Count: 0

ID	Name	Total Requests	Total Requested MIPS

ID	Application ID	Servers	Available MIPS	Difference

ID	MIPS
1	35000
2	35000
3	35000
4	35000
5	35000

.\extendedevents.xml

Figure 4.6: Full datacentre GUI

In a script, there are events which refer to a simulated occurrence of anything the script can simulate at a specified point in time. A single event could be the addition or removal of an application into the datacentre or it could a set of simulated requests for a specified application. Individual events must be described in a script via an event type. This tells the parser how to read, generate and process the content within the event type element. Four different event types can be used in a script to automatically generate requests or to add / remove an application for the simulation. This is the format description for the event script elements:

- Add/RemoveAppEvent

  AppName - The name of the application to add or remove from the datacentre model

  Offset - The desired time for the event to occur, offset from the start of the simulation run

- RandomEvents - Produces total request sizes, randomly, between a minimum and maximum size, for a specified time period

  AppName - The name of the application this script is targeting

  StartOffset - The desired time for the events to start, offset from the start of the simulation run

  EndOffset - The desired time for the events to end, offset from the start of the simulation run

  RequestSize - The request size of a single request

  Frequency - How often, between the start and end offset, an event should occur

  MinTotal - The minimum random total request size (this is a multiple of RequestSize as near to MinTotal as possible)

  MaxTotal - The maximum random total request size (this is a multiple of RequestSize as near to MaxTotal as possible)

- LinearEvents - Produces request sizes that linearly increase or decrease between an initial and final size for the specified time period. LinearEvents has the same elements as for RandomEvents but without Min/MaxTotal. In addition:

  InitialSize - The total request size to produce at the StartOffset

FinalSize - The total request size to produce at the EndOffset (request sizes between these two value should steadily increase or decrease over the time period)

- BandedEvents - The same as for LinearEvents but the request total is random within a set percentage from the linear

  PlusMinusBand - A percentage that the total request size is permitted to deviate randomly to (from the normal linear value)

Following is a typical but simple xml event script showing these elements, it simply adds four applications and removes one of them. It then applies the three request event type elements to two of the applications:

```
<EventScript>
  <AddAppEvent>
    <AppName>app1</AppName>
    <Offset>0</Offset>
  </AddAppEvent>
  <AddAppEvent>
    <AppName>app2</AppName>
    <Offset>0</Offset>
  </AddAppEvent>
  <AddAppEvent>
    <AppName>app3</AppName>
    <Offset>0</Offset>
  </AddAppEvent>
  <RemoveAppEvent>
    <AppName>app2</AppName>
    <Offset>1</Offset>
  </RemoveAppEvent>
  <RandomEvents>
    <AppName>app1</AppName>
    <StartOffset>1000</StartOffset>
    <EndOffset>30000</EndOffset>
    <RequestSize>15000</RequestSize>
    <Frequency>2000</Frequency>
    <MinTotal>4000</MinTotal>
    <MaxTotal>100000</MaxTotal>
  </RandomEvents>
  <LinearEvents>
    <AppName>app1</AppName>
    <StartOffset>30000</StartOffset>
    <EndOffset>60000</EndOffset>
    <RequestSize>15000</RequestSize>
    <Frequency>2000</Frequency>
    <InitialSize>15000</InitialSize>
    <FinalSize>600000</FinalSize>
  </LinearEvents>
  <BandedEvents>
    <AppName>app3</AppName>
    <StartOffset>1000</StartOffset>
    <EndOffset>60000</EndOffset>
    <RequestSize>15000</RequestSize>
    <Frequency>2000</Frequency>
    <InitialSize>150000</InitialSize>
    <FinalSize>400000</FinalSize>
    <PlusMinusBand>20</PlusMinusBand>
  </BandedEvents>
</EventScript>
```

To illustrate the event types (RandomEvents, LinearEvents and BandedEvents) better, 3 tests were performed and the request size vs. time is graphed. In each test there is only one application using one of the three event types where the min/maximum size or initial/final size is set to 15000 and 600000 respectively. Start and End offset is 1 and 60 seconds respectively. Frequency is set to every second and for the banded test the PlusMinusBand is set to 20 %. Figure 4.7 shows the graphed outcome of the tests.

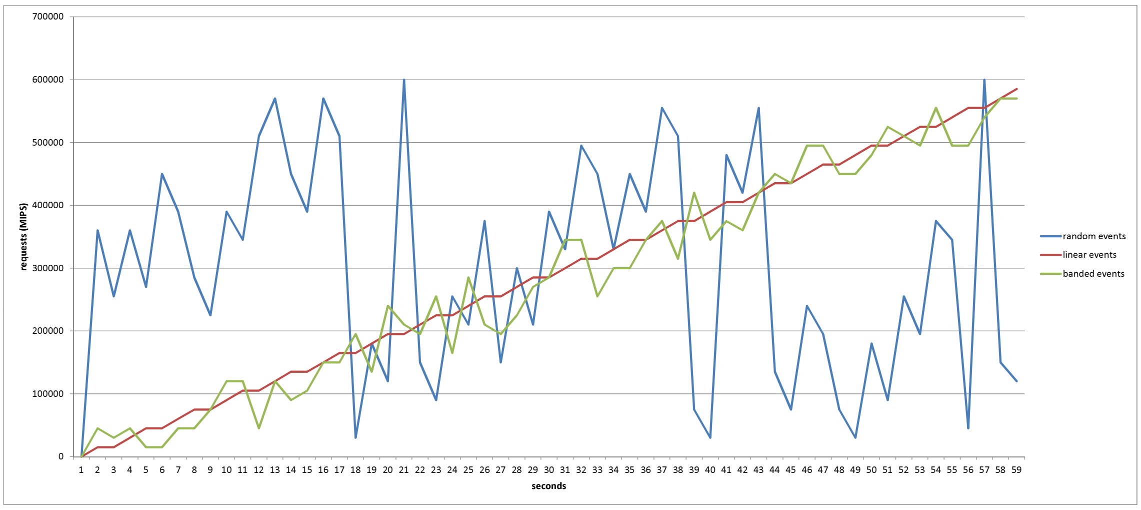


Figure 4.7: Graphical representation of request generator script event types

This graph is the expected output of the test runs and as such confirms and illustrates the script functions.

4.2.4 Composing the datacentre T-R Manager through TAP

Given the difficulties in composing a T-R program, raw and without any harness this section develops a basic T-R manager for the datacentre problem domain using our TAP mechanism. It shows how errors and other issues are easily highlighted using TAP. We can approach the composition of the T-R manager with a 'divide and conquer' type approach. i.e. begin with a very small manager, use the TAP mechanism to find problems, fix the problems by adding or editing the rules then repeat the process. In this way we make small incremental changes to our design whilst verifying each stage with TAP.

The initial manager should be complete, i.e contains no errors and is capable of reaching the goal from any given state but will not attempt any extended capabilities at this stage.

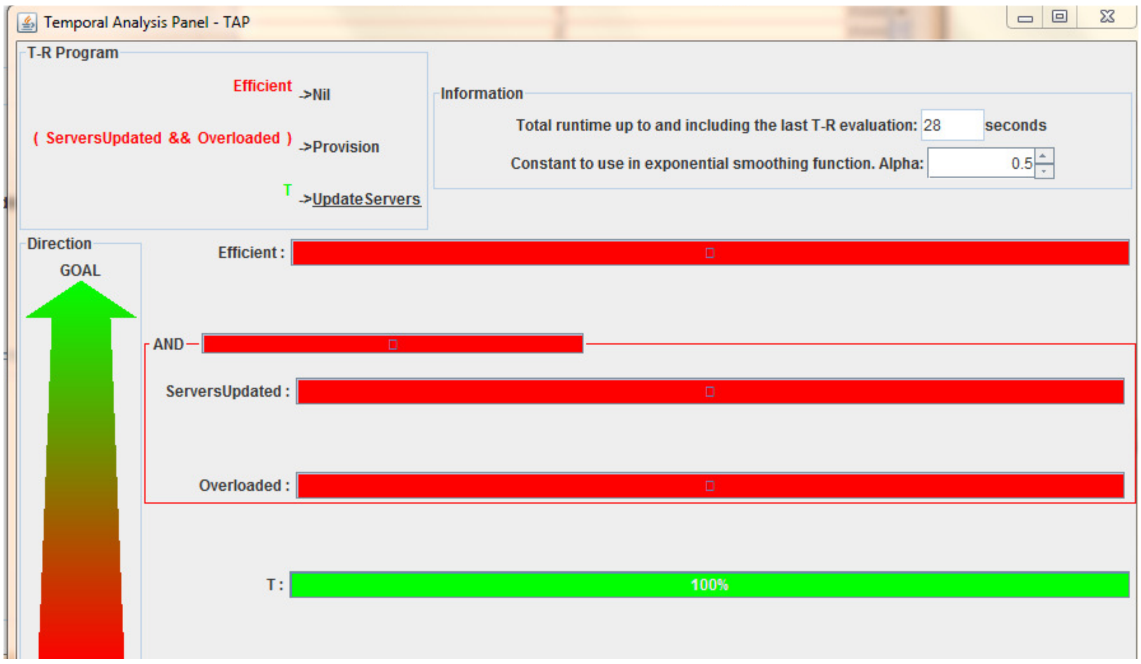
The objective of the datacentre would be to quickly satisfy the needs of all the requests without much unnecessary wastage. In other words, each application served by the datacentre should be allocated enough servers in relation to the amount of present requests for it, otherwise the users of the application would notice a distinct degradation in performance. Furthermore, each application pool should not contain more servers than is necessary as this would prevent other pools utilising the underloaded server. This objective is captured as a single conditional statement which can then be set as the managers' goal. So if all pools are not overloaded and not underloaded we can call this condition 'Efficient'.

When servers are provisioned they are moved first to a provisioning queue as each server will take a set amount of time to prepare. We will need a 'Provision' action to move idle servers to the queue as well as an action to move servers in the provisioning queue to their desired pool where the provision time for the server has expired, this action we will call 'UpdateServers'. This can then become our initial action since no prerequisite states are necessary for it to run. We also need 'UpdateServers' to occasionally execute or no server will ever reach its destination pool, however, rules to deal with conditions where a pool is overloaded for example should take priority.

The manager cannot reach the 'Efficient' goal directly from 'UpdateServers' because there will not be any servers in the provision queue to update without a 'Provision' action. The 'Provision' requires that we only provision a server to a pool if that pool needs extra servers to run efficiently so we should add the condition 'Overloaded' to query whether there are any pools which have more requests than the current servers can cope with.

We also realize that when a pool becomes overloaded it will begin to provision servers, but those servers will never reach their destination pool because the 'Provision' rule takes priority over 'UpdateServers' and while the pool is overloaded 'UpdateServers' can never execute.

To fix this problem we can create a condition to check if there are any servers which need to be moved to a pool from the provisioning queue. We then logically join the condition to 'Overloaded', ensuring that servers are correctly updated if needed before any new provision takes place. This condition compliments 'UpdateServers' and is called 'ServersUpdated'.



```
Efficient ->Nil
( ServersUpdated && Overloaded ) ->Provision
T ->UpdateServers
```

Total runtime up to and including the last T-R evaluation: 28 seconds

Constant to use in exponential smoothing function. Alpha: 0.5

Efficient :

ServersUpdated :

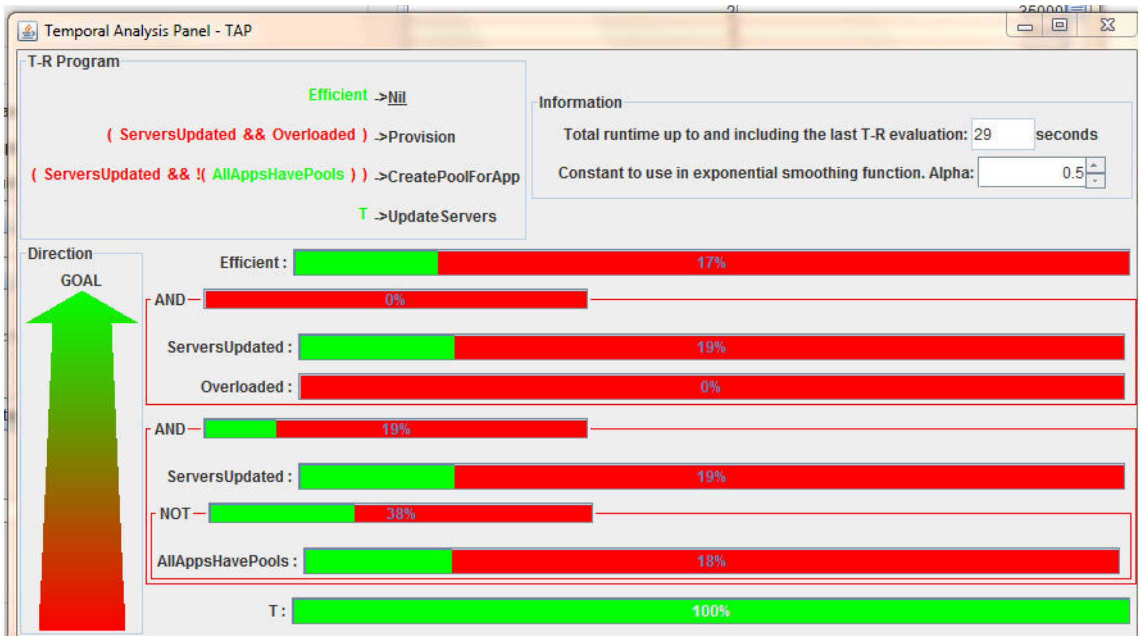
Overloaded :

T : 100%

Figure 4.8: Basic manager: First iteration.

When this program is executed, TAP reveals that 'Overloaded' remains on 0% for the duration meaning it never became true. After inspection, it was revealed that because 'Overloaded' inspects pools for overloads and no pool is ever created in our current T-R program then it is impossible for 'Overloaded' to become true.

A rule will be needed to add a pool for each application in the datacentre. This is shown in figure 4.9.


Temporal Analysis Panel - TAP

T-R Program

Efficient ->Nil
(ServersUpdated && Overloaded) ->Provision
(ServersUpdated && !(AllAppsHavePools)) ->CreatePoolForApp
T ->UpdateServers

Information

Total runtime up to and including the last T-R evaluation: 29 seconds
Constant to use in exponential smoothing function. Alpha: 0.5

Direction

GOAL

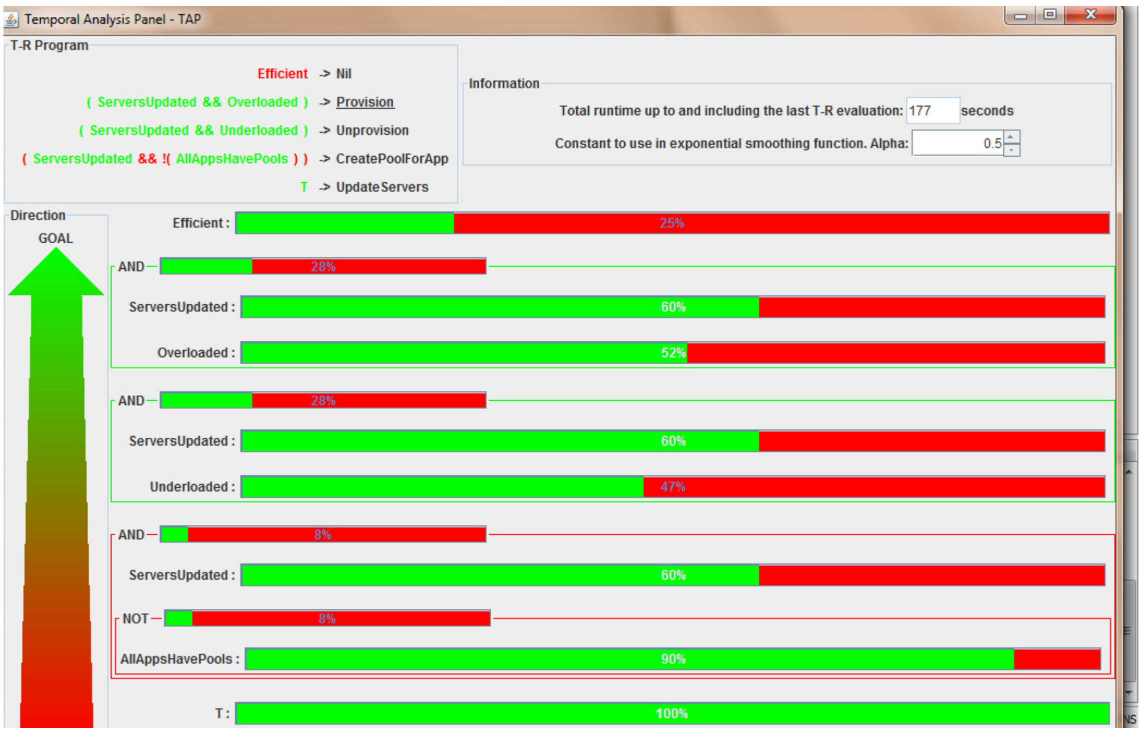
Efficient : 17%
AND 0%
ServersUpdated : 19%
Overloaded : 0%
AND 19%
ServersUpdated : 19%
NOT 38%
AllAppsHavePools : 18%
T : 100%

Figure 4.9: Basic manager: Second iteration

The program can now achieve its goal of becoming efficient, however, for a finite number of available servers and assuming pools will require a greater capacity for requests than is currently available at some moment in time then the overloaded pool will remain overloaded if all the servers have been allocated elsewhere, even if there are pools which are underutilising their servers. TAP confirms the problems existence when the Efficient goal true proportion returns to 0% when all idle servers have been distributed.

We need to return these underutilised servers to an idle state so that they can be provisioned to overloaded pools. We therefore add another rule for this purpose. This rule will contain a condition to check for pools with too many servers (‘Underloaded’) and an action to return these servers to an idle state (‘Unprovision’). The result of adding this rule is shown in figure 4.10.

Before a simulation begins, the number of idle servers can be chosen. If the number of idle servers are reduced and the simulation is run, we can see that ‘Overloaded’ quickly becomes true, which is briefly returned to false as idle servers are



Temporal Analysis Panel - TAP

T-R Program

```
                                      Efficient  -> Nil
          ( ServersUpdated && Overloaded )  -> Provision
         ( ServersUpdated && Underloaded )  -> Unprovision
( ServersUpdated && !( AllAppsHavePools ) ) -> CreatePoolForApp
                                          T  -> UpdateServers
```

Information

Total runtime up to and including the last T-R evaluation: 177 seconds

Constant to use in exponential smoothing function. Alpha: 0.5

Direction

GOAL

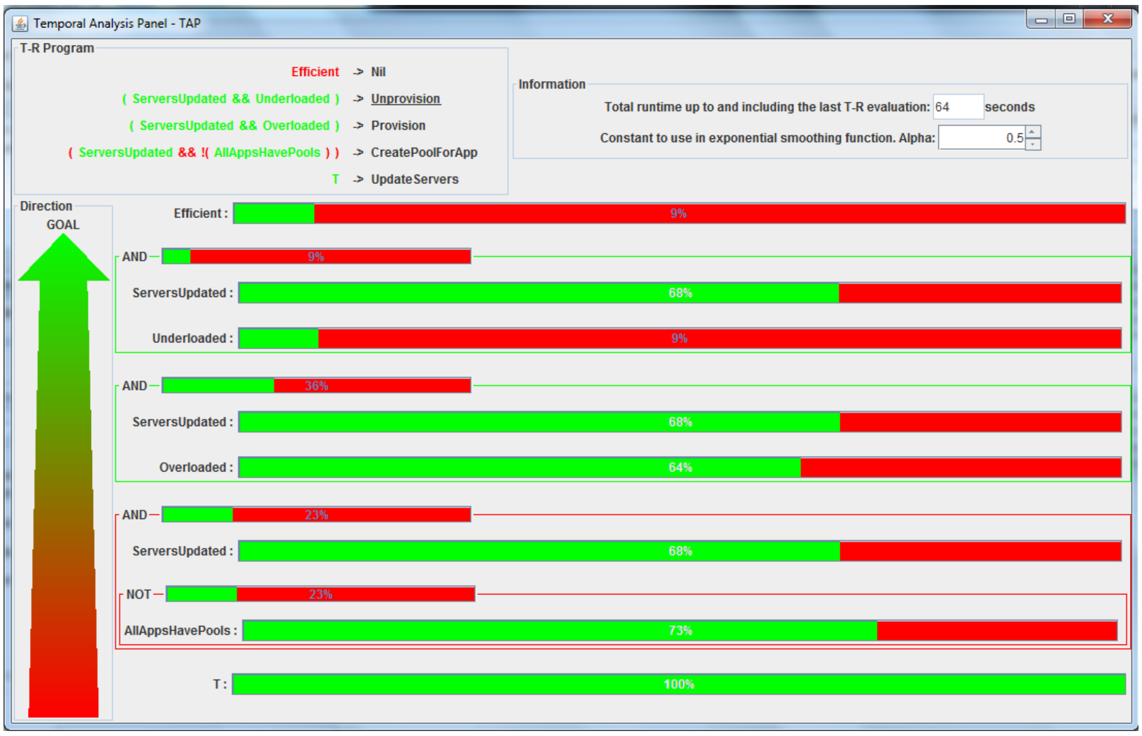
```
Efficient : 25%
AND 28%
    ServersUpdated : 60%
    Overloaded : 52%
AND 28%
    ServersUpdated : 60%
    Underloaded : 47%
AND 8%
    ServersUpdated : 60%
    NOT 8%
        AllAppsHavePools : 90%
T : 100%
```

Figure 4.10: Basic manager: Third iteration

distributed. Whenever the number of requests is reduced for a pool for a time, ‘Underloaded’ becomes true, however, if ‘Overloaded’ becomes true concurrently then servers cannot be removed from underloaded pools because the ‘Provision’ rule currently take priority over the ‘Unprovision’ rule. There are no more idle servers which can be provisioned to the overloaded pool either so we now have a stalemate situation where some pools need servers but cannot get any, and some pools have servers which are not needed but they cannot be removed. TAP alerted us to this problem when the ‘Efficient’ true proportion quickly reduced to 0% and remained on this value.

This problem could be resolved by giving priority to the ‘Unprovision’ rule. In this way, all servers are returned to idle state ready to be utilised before pools receive servers when needed. Figure 4.11 shows the ‘Unprovision’ rule given priority over ‘Provision’. This simple change ensures the correct operation of the datacentre manager.

Building the datacentre with the use of the temporal analysis tool demonstrates


Temporal Analysis Panel - TAP

T-R Program

```
                                    Efficient  -> Nil
          ( ServersUpdated && Underloaded )  -> Unprovision
           ( ServersUpdated && Overloaded )  -> Provision
( ServersUpdated && !( AllAppsHavePools ) )  -> CreatePoolForApp
                                            T  -> UpdateServers
```

Information

Total runtime up to and including the last T-R evaluation: 64 seconds

Constant to use in exponential smoothing function. Alpha: 0.5

Figure 4.11: Basic manager: Final version

that although software built using the T-R paradigm reduces human maintenance significantly, it is also very easy to introduce logical errors. This basic construction example shows how TAP can reduce the time needed for the discovery of these errors. TAP is therefore an excellent tool for the location of logical validation problems and well as verification. This is significant for the evolution of T-R programs since issues here are often include the fact that a T-R program does not behave in the way the developer intended and less about errors in code.

4.2.5 Extensions of the basic manager

The developer of the datacentre or any T-R program will probably need or want to improve the first iteration of the software. For example, the basic datacentre manager successfully completes its main task of distributing idle servers to the applications which need them in order to improve request response time for those applications, but there are many improvements which could be made (this process of requiring additional improvements over time would occur in any real-world system). The basic manager is extended to include the new requirements and through JTRAF we can use the automatic logging of TAP to compare one manager against another. We can ultimately conclude on whether a new version helps or hinders the situation.

In this section we build two extensions to the basic datacentre manager and compare all three through TAP logger. In the first extension the servers are more evenly distributed to the applications that require them rather than taking the basic manager approach of provisioning servers to the first pool in need. This will hopefully avoid the situation where some pools have no or very few servers. The second extension, will attempt to analyse the trend of datacentre usage and distribute servers in advance of them being required.

Sub-T-R extension

In the basic manager, when the datacentre becomes overloaded the provision action executes. This simply provisions an idle server to the next available overloaded pool. A more beneficial approach would be to provision a server to the 'most overloaded' pool first otherwise pool 1 could receive many servers whereas other pools may be allocated none. Also, if there are no idle servers available it might be beneficial to remove a server from a pool and return it to an idle state before provisioning the server to the 'most overloaded' pool. This ensures a fair distribution of servers and is exactly the method employed by this advanced manager program.

Because the described process is more complex than the provision action used in the basic manager, we have created a separate T-R program for this purpose as shown in figure 4.12. This is then called by the main program of figure 4.5 in place of the single provision action of the basic manager.

A sub-T-R program is simply a T-R program which is called by another, where all T-R programs are types of Actions (see figure 3.2). Actions and TRPrograms can then be used interchangeably. The advantage being that more complex problems which demand several steps to solve can take the form of (and gain all the

$$\begin{aligned}
\neg Overloaded &\longrightarrow Nil \\
HasIdle \wedge GotMostUrgent &\longrightarrow MoveIdle \\
(\neg HasIdle \wedge GotMostUrgent \wedge & \\
GotLeastOverloaded \wedge ValidUnprovision) &\longrightarrow Unprovision \\
\neg HasIdle \wedge GotMostUrgent &\longrightarrow GetLeastOverloaded \\
T &\longrightarrow GetMostUrgent
\end{aligned}$$

Figure 4.12: Datacentre sub-T-R program

advantages of) complete T-R programs. This leads to more extensible and reusable code. As with individual rules, the calling T-R program takes precedence over the called T-R program. So, if a sub-T-R program is currently running but a higher precedence rule in the calling program becomes true, then the called T-R program will immediately cease execution and control passes back to the calling program. The sub-T-R program behaves exactly like any other action in this way.

This approach is used in the datacentre and tests were performed on two versions of the data-centre manager. The basic version uses 'Provision' as a single action which simply assigns an idle server to the nearest overloaded pool. The second version replaces the action 'Provision' with a complete T-R program, which uses context information to both assign a server to the most 'deserving' of pools first, and more fairly distribute the already provisioned servers if no idle ones exist.

To show the difference between these two versions, some tests were performed showing the difference in the capacity (number of MIPS the pool is capable of serving vs. the number MIPS being requested) of each pool. The basic manager (using a single Action to provision) is referred to in the tests as 'Provisioned' and the advanced version (using a sub-T-R program in place of the 'Provisioned' action) is referred to as 'Sub-TR Provisioned'.

In each test there are four applications. Each application has a starting value for the number of MIPS requested and a new value of requested MIPS. Recorded, is the actual number of MIPS provided and the difference between this actual value and the requested one for both versions of the manager.

In every test, there are 60 servers of 35000 capacity each. This means there is a total of 2100000 MIPS to share between four applications. If the request total is higher, the idle server list will be empty and the data-centre is overloaded. Tables 4.1 - 4.4 show these differences.

Table 4.1: Application 1 test scenarios. Values taken from Data-centre with and without sub-TR version

Starting request	New request	Provisioned	Difference	Sub-TR provisioned	Difference
570000	630000	630000	0	595000	-35000
0	630000	630000	0	595000	-35000
630000	810000	630000	-180000	630000	-180000
810000	450000	455000	5000	455000	5000
450000	450000	455000	5000	420000	-30000
450000	210000	210000	0	210000	0
210000	90000	105000	15000	105000	15000
90000	870000	875000	5000	665000	-205000

Table 4.2: Application 2 test scenarios. Values taken from Data-centre with and without sub-TR version

Starting request	New request	Provisioned	Difference	Sub-TR provisioned	Difference
420000	465000	420000	-45000	455000	-10000
0	465000	490000	25000	455000	-10000
465000	600000	490000	-110000	420000	-180000
600000	330000	350000	20000	350000	20000
330000	375000	385000	10000	315000	-60000
375000	285000	315000	30000	315000	30000
285000	120000	140000	20000	140000	20000
120000	510000	525000	15000	315000	-195000

Table 4.3: Application 3 test scenarios. Values taken from Data-centre with and without sub-TR version

Starting request	New request	Provisioned	Difference	Sub-TR provisioned	Difference
720000	795000	735000	-60000	770000	-25000
0	795000	805000	10000	770000	-25000
795000	1020000	805000	-215000	840000	-180000
1020000	870000	875000	5000	875000	5000
870000	1095000	910000	-185000	1050000	-45000
1095000	945000	945000	0	945000	0
945000	120000	140000	20000	140000	20000
120000	1170000	700000	-470000	945000	-225000

The graphs of 4.13 show the tables in graphical form. They highlight the differences between the two manager versions. Where a bar is close to zero, the difference

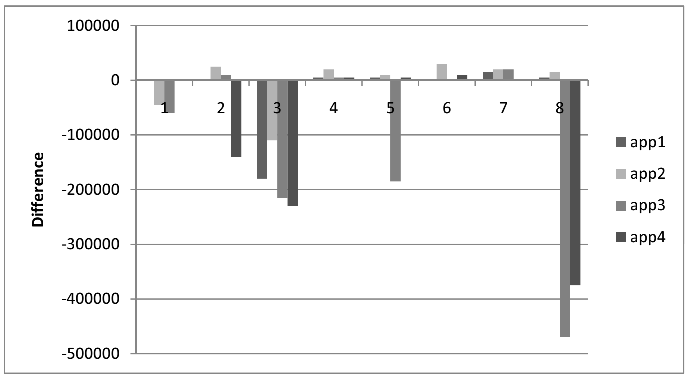
Table 4.4: Application 4 test scenarios. Values taken from Data-centre with and without sub-TR version

Starting request	New request	Provisioned	Difference	Sub-TR provisioned	Difference
285000	315000	315000	0	280000	-35000
0	315000	175000	-140000	280000	-35000
315000	405000	175000	-230000	210000	-195000
405000	345000	350000	5000	350000	5000
345000	345000	350000	5000	315000	-30000
345000	375000	385000	10000	385000	10000
375000	0	0	0	0	0
0	375000	0	-375000	175000	-200000

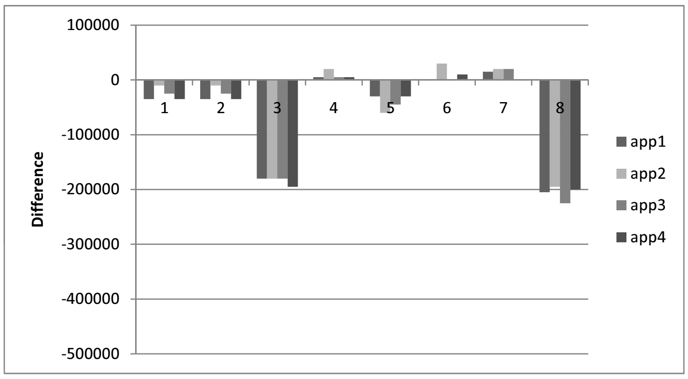
with respect to requested and provisioned MIPS is low and is therefore very efficient. When a bar is much greater than or much less than zero, the pool/application is underloaded or overloaded respectivly and is very inefficient. In figure 4.13(a) it is shown that the simpler manager version does not fairly distribute the servers when an overloaded situation occurs. In fact, this version simply provides an idle server to the next available application or pool. The second version distributes and moves servers to the 'most overloaded' pool/application first. It can clearly be seen in figure 4.13(b) that the servers are more evenly distributed across each application.

This effect is most prominent at stage eight, where although app1 and app2 of figure 4.13(a) are efficient, app3 and app4 are are extremely inefficient. Whereas with the sub-T-R program in place and in figure 4.13(b) we can see that at stage eight all the applications are inefficient but none of the applications are inefficient to the same degree as with the basic version. This may or may not be desired depending on the system requirements of course.

The purpose of this extension is to show the simplicity of extending a T-R program in this way and thus demonstrates the extensibility of T-R. It is also possible that the developer may wish to give priority to one or more applications. In which case additional extensions could be made for this purpose, perhaps using high-level policy scripts or utility functions.



(a) Difference between requested MIPS and provisioned MIPS for applications in each test scenario where provision is a single action



(b) Difference between requested MIPS and provisioned MIPS for applications in each test scenario where provision is a sub-TR program

Figure 4.13: Server distribution differences

Trend analysis manager

We have built a T-R manager based on the idea of using trend analysis to predict the increase and decrease in request levels and provision servers to pools before they become overloaded. As the provisioning of servers is not instant and a server could be held in the provisioning queue for up to four minutes depending on the simulation settings, it seems trend-analysis approach could be an ideal method of provisioning servers before a pool requests the servers, and could improve overall request to response time. This trend analysis technique is not designed to rival existing autonomic techniques however but shows how other techniques can easily be employed within a T-R program. We can use TAP to observe the effectiveness of this manager compared with the other managers.

The exponential smoothing function used in TAP can be used to forecast trends in the data and possibly improve the datacentre efficiency by increasing pool capacity before the need to provision servers, hence a new manager was constructed for this purpose. The basic exponential smoothing formulae used in this manager is:

$$S_t = \alpha y_{t-1} + (1 - \alpha)S_{t-1}$$

where S_t is the smoothed term, $_t$ is the time period, y is the measured value and α is a value between 0 and 1. Where a higher value for α gives a higher weight to more recent values and thus less of a smoothing effect, a smaller value results in the opposite (less weight to more recent results but a greater smoothing effect).

We had originally thought that an extensive sub-T-R extension using several rules would be needed to implement this trend analysis manager. However, we only needed to make changes to two conditions, ‘underloaded’ and ‘overloaded’. We should return true from these conditions if a pool/application is *predicted* to become underloaded or overloaded at some moment in the future rather than returning true if a pool/application is currently underloaded or overloaded as is currently the case. We need to investigate some values for an ideal constant to use for α with respect to how close the predicted value is from the real one (error) and we need to know the rate that we take samples.

There are many values which could be varied and/or different datasets could provide different results for our simulation and it would be impossible to investigate all these possibilities therefore the experiments shown could undoubtedly be improved to provide an increase in datacentre simulation efficiency. The datacentre is however only the scenario used for our T-R experiments, we are not investigating datacentres themselves. For example, the ideal smoothing constant under investigation depends largely on server provision time (currently thirty seconds), T-R program evaluation delay time (current one second), the frequency of change request (default script is every 6 minutes) and the size and rate of change of incoming requests.

The remainder of this chapter will investigate values to give an effective trend analysis version of the datacentre T-R manager. We will do this by first running a suitable simulation and logging the results. Since we want to predict if a pool needs an increase and decrease in the capacity the data we log is of the difference between available MIPs and requested MIPs. This will give us a positive or negative value depending on whether a pool needs to is above or below capacity.

The script used in the test automatically generated four applications and then proceeded to generate banded event request inputs (see section 4.2.3) for each application every minute for a total of thirty six minutes. The number of requests generated increased and decreased at various gradients every six minutes. This data was logged to a file and graphs were generated.

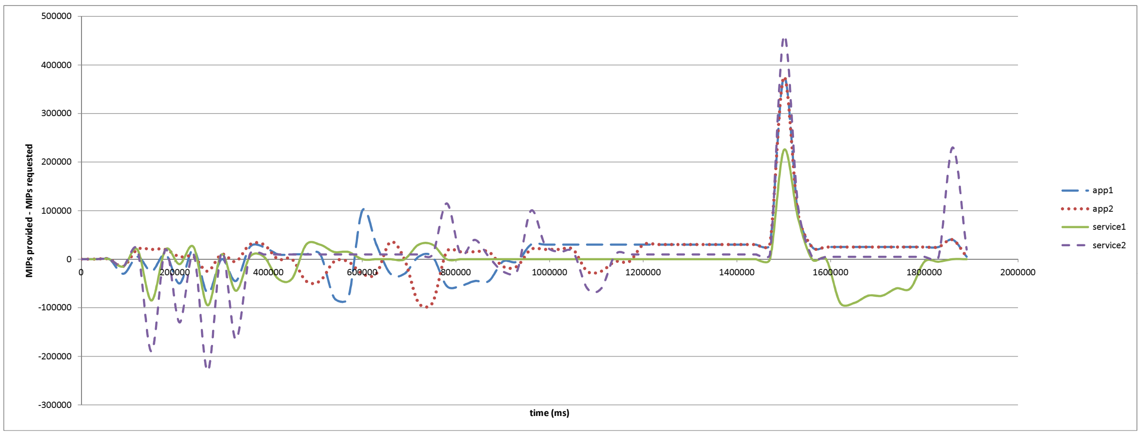


Figure 4.14: Pool capacity - requests vs. time: Thirty minute simulation run using four applications

Figure 4.14 is the result of logging the described script for roughly thirty minutes. The graph lines represent the difference between the capacity of each application pool and the actual number of requests for each application. This means that if the line is below zero (a negative value) the pool is overloaded and requires more capacity (extra servers), if the line is above zero (a positive value) the pool has too much capacity and servers can be safely returned to idle state.

The question we need answering is can we use exponential smoothing on the logged data to predict future trends and improve the efficiency of the datacentre?

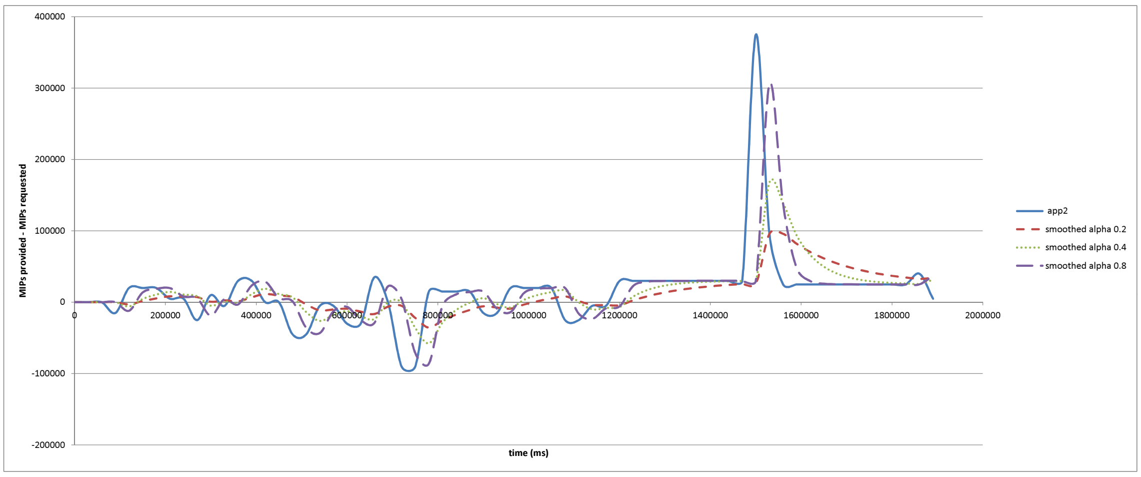


Figure 4.15: Calibration of smoothing constant: 'app2' data from figure 4.14 with three forecast datasets (three values for α; 0.2, 0.4, 0.8)

In figure 4.15 the solid line represents only 'app2' from the figure 4.14 (the real observed measurements). The other lines show exponentially smoothed results of 'app2' using three different values for α, 0.2, 0.4 and 0.8. We are comparing the smoothed results with the real measurements in order to gain some indication as to which smoothing constant would be effective at predicting trends and thus improving efficiency.

We observe that a higher α value produces a line which more closely matches the real data but is less smooth than smaller α values. In theory, closely matched smoothed data would be less effective because these values differ very little from the observed measurements, and because these values are prediction of future events, changes made to the datacentre would be made after a sudden change and would possibly have a negative impact on effectiveness. A smaller α value produces less impact to its results when a sudden change is observed meaning the datacentre would be underloaded or overloaded by a large margin of error but only for a short period of time.

In chapter 5 we will perform experiments and compare the use cases presented here. For the datacentre, the three T-R managers will manage the datacentre and their efficiency will be compared using the generated log file from TAP. The results should reveal whether the trend analysis version of the T-R manager reduced or improved efficiency over the other managers. This could be used to make improvements to any scenario using JRTAF as each condition of a T-R manager is automatically logged.

CHAPTER 5

EMPIRICAL INVESTIGATION

The two use-cases here explore the use of T-R programs as managers for autonomic systems. The T-R mechanism is explored as intended (without the use of any additional components or extensions), showing the benefits of the system to autonomics. We then explain how the extensions made to JTRAF can be used for diagnosing errors and comparing and quantifying changes made. The summary section describes the investigations with regards to what has been done and what conclusions we can arrive at about T-R in autonomics.

The first example is a simple file sending case which partly mimics the TCP protocol. It demonstrates how a T-R system works, can be applied to other similar projects, and explains the basic ideas of T-R without the distractions of the extensions built into JTRAF. The second example shows how T-R and JTRAF can be used to manage a datacentre. There are three managers constructed for managing the same datacentre model. The graphical comparisons demonstrate the use of the JTRAF extensions in providing data required for comparisons and improvements to be made as well as their use as a diagnostic and debugging mechanism.

5.1 File Sender

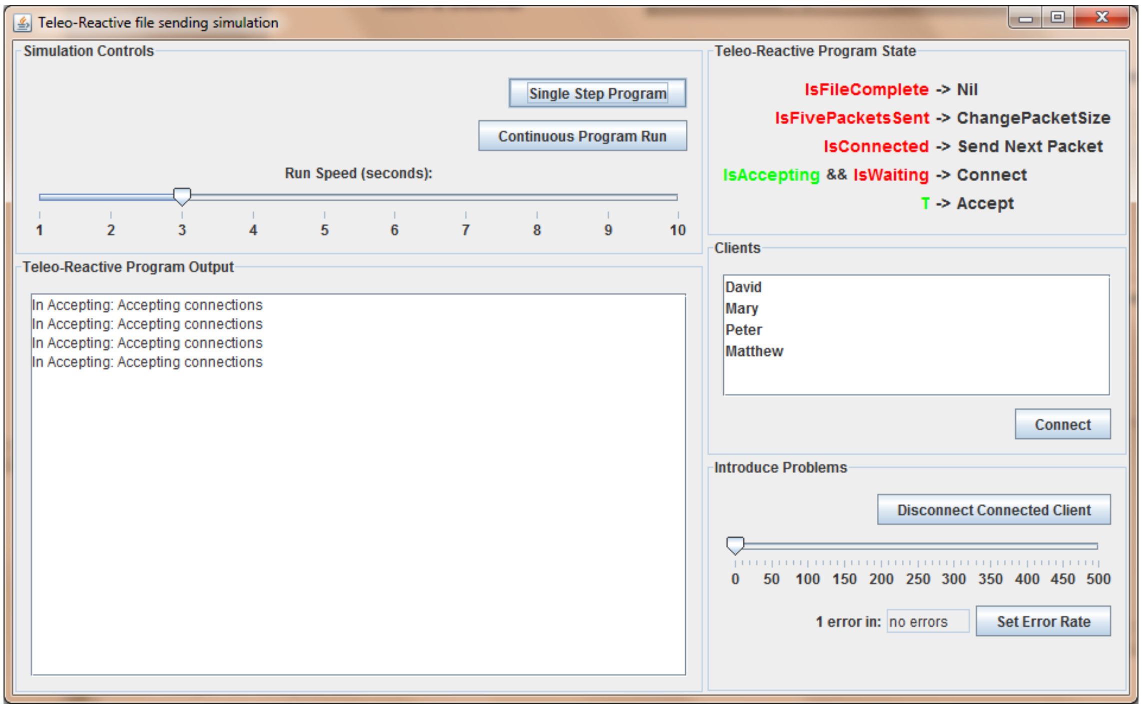
The file sending demonstrator sends a single file across a network to a connected client. The file is sent as separate packets with the assumption that the client program can reassemble the packets as they are received. The application is a hybrid of the TCP protocol and a simple file sending application but the simulation here is not meant to improve or replace any existing systems but is purely aimed at demonstrating the T-R approach.

The demonstrator in section 4.1 describes the file sender T-R manager and controls in detail so, after being initially presented with the program, and assuming no client has connected, the only true condition is ‘T’ and so on first evaluation, the program will execute the initial action ‘Accept’ as shown in figure 5.1. The ‘Accept’ action changes ‘IsAccepting’ from false to true but the subsequent action cannot execute until a client enters the queue and ‘IsWaiting’ becomes true. ‘Accept’ will continually execute upon each evaluation (effectively forming a loop) which is quite simple to initiate if this program was constructed in any language, however, T-R grants the power to very easily move between rules at any point in time. If at a later point, there was an error or unexpected event the program would intuitively move to this initial state and begin waiting for a client again. This ‘rollback’ mechanism is often not so easy to implement in a standard programming language and would not be so robust.

5.1.1 Self-Healing Behaviour

To better illustrate this point we can demonstrate the effects of a disconnection. Moving forward in time to a point where a client ‘Mary’ is connected and several packets of the file have been sent (see figure 5.2) What will happen if Mary becomes suddenly disconnected?

If the client becomes disconnected the condition ‘IsConnected’ becomes false. The conditions ‘IsAccepting’ and ‘IsWaiting’ are also false so this rule cannot fire



```
IsFileComplete -> Nil
IsFivePacketsSent -> ChangePacketSize
IsConnected -> Send Next Packet
IsAccepting && IsWaiting -> Connect
T -> Accept
```

```
In Accepting: Accepting connections
In Accepting: Accepting connections
In Accepting: Accepting connections
In Accepting: Accepting connections
```

David
Mary
Peter
Matthew

1 error in: no errors

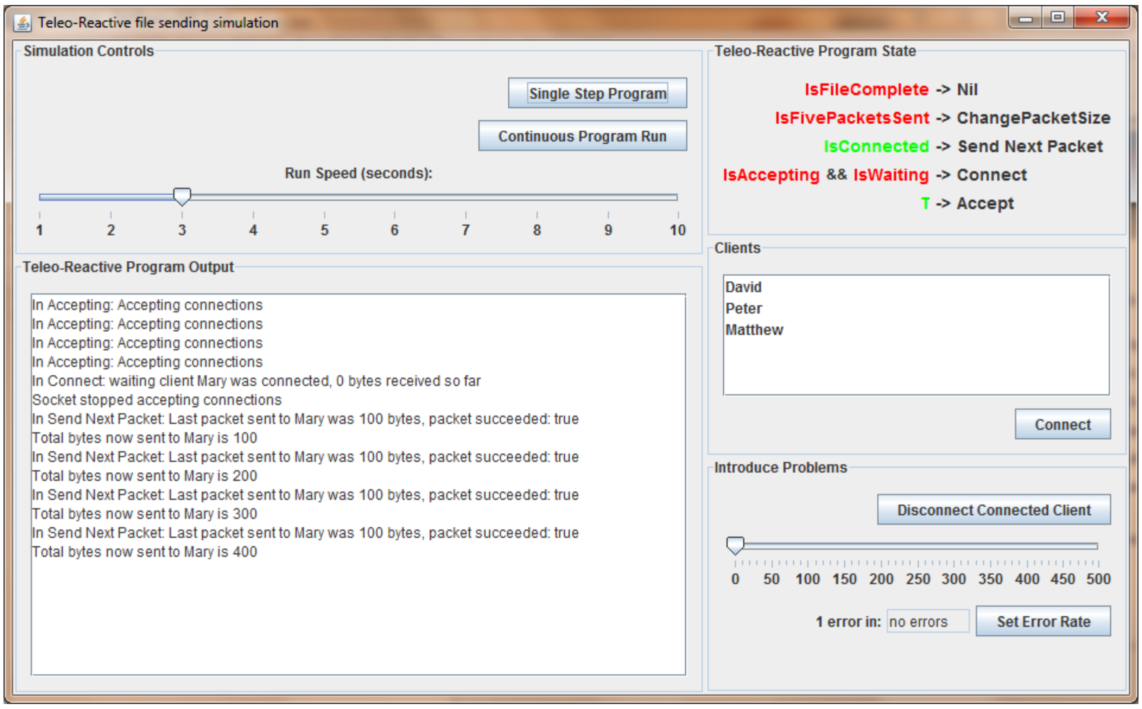
Figure 5.1: Initial state of the file sender application after first evaluation

either. The highest true condition is 'T' so the program is forced to return to the initial rule and the original state as shown in figure 5.3.

The program will continue indefinitely from this initial rule until a higher condition becomes true. If it is possible for Mary to reconnect then she will enter the waiting queue. The conditions of the 'Connect' action are now true so this action can fire. 'IsConnected' is true again so packets resume sending once more. Eventually the goal 'IsFileSent' will become true and the program will be complete. If the client disconnects again then this process of recovery will start again.

What is important here is that a properly constructed T-R program should never fail outright, it will simply fall through to a lower priority rule if an error occurs and continue from that point. We also never needed to specify how the client disconnected despite the many reasons for the failure, from a server issue to a physical cable disconnection. All that the T-R program needs to know is whether 'IsConnected' is true or false.

This circumvention or recovery from the fault gives a T-R program its inherent robustness and is present in any properly constructed T-R program. Whilst the code



```
IsFileComplete -> Nil
IsFivePacketsSent -> ChangePacketSize
IsConnected -> Send Next Packet
IsAccepting && IsWaiting -> Connect
T -> Accept
```

```
In Accepting: Accepting connections
In Accepting: Accepting connections
In Accepting: Accepting connections
In Accepting: Accepting connections
In Connect: waiting client Mary was connected, 0 bytes received so far
Socket stopped accepting connections
In Send Next Packet: Last packet sent to Mary was 100 bytes, packet succeeded: true
Total bytes now sent to Mary is 100
In Send Next Packet: Last packet sent to Mary was 100 bytes, packet succeeded: true
Total bytes now sent to Mary is 200
In Send Next Packet: Last packet sent to Mary was 100 bytes, packet succeeded: true
Total bytes now sent to Mary is 300
In Send Next Packet: Last packet sent to Mary was 100 bytes, packet succeeded: true
Total bytes now sent to Mary is 400
```

Clients: David, Peter, Matthew

1 error in: no errors

Figure 5.2: A client has connected to the File sending service and four packets have been sent

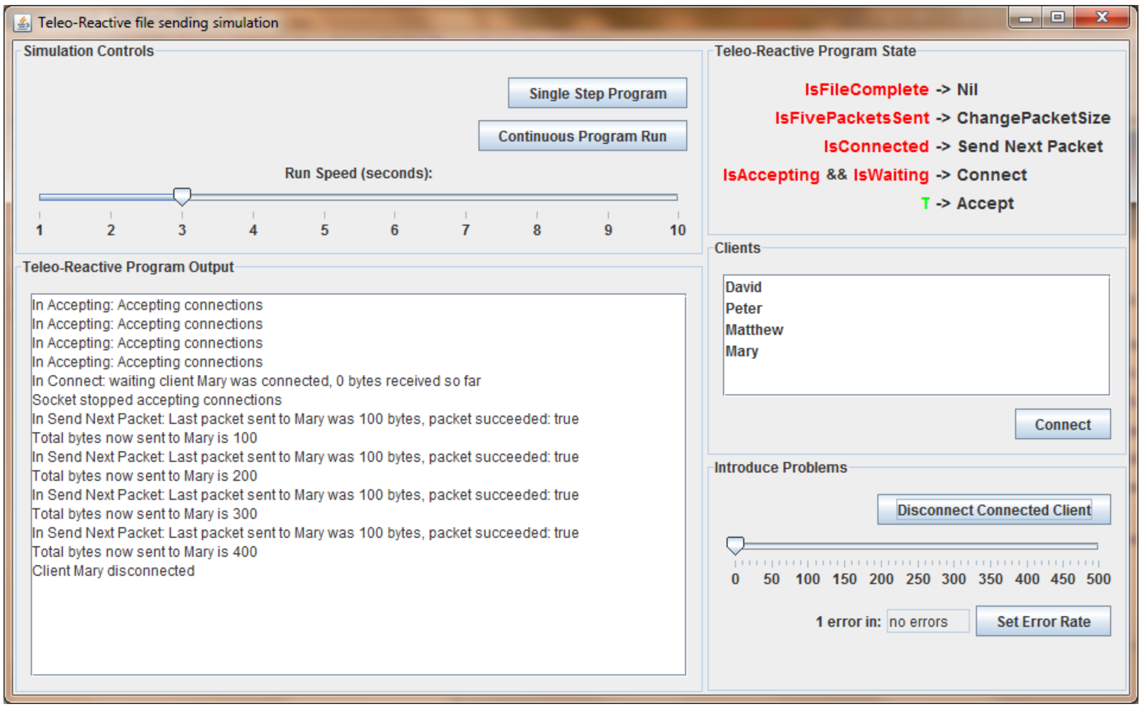
itself does not dynamically alter, the T-R program is entirely runtime adaptable, makes changes to suit the current circumstances and could be termed self-healing.

5.1.2 Self-Optimising Behaviour

It was quite simple to add a single rule to the file sending example to alter the packet size according to recent communication quality history and thus adding self-optimising behaviour. If the communication quality drops and very few packets are transferred with any success the the packet size is reduced to compensate. If the quality improves then the packet size is increased.

The rule is quite simple, the condition 'IsFivePacketsSent' is true if a multiple of five packets has been transferred. The action 'ChangePacketSize' inspects the last five packets sent and adjusts the size accordingly.

To demonstrate this a client is connected as before and the error rate is increased. We send five packets and observe what happens in the T-R program state. Figure 5.4 shows this.



```
IsFileComplete -> Nil
IsFivePacketsSent -> ChangePacketSize
IsConnected -> Send Next Packet
IsAccepting && IsWaiting -> Connect
T -> Accept
```

```
In Accepting: Accepting connections
In Accepting: Accepting connections
In Accepting: Accepting connections
In Accepting: Accepting connections
In Connect: waiting client Mary was connected, 0 bytes received so far
Socket stopped accepting connections
In Send Next Packet: Last packet sent to Mary was 100 bytes, packet succeeded: true
Total bytes now sent to Mary is 100
In Send Next Packet: Last packet sent to Mary was 100 bytes, packet succeeded: true
Total bytes now sent to Mary is 200
In Send Next Packet: Last packet sent to Mary was 100 bytes, packet succeeded: true
Total bytes now sent to Mary is 300
In Send Next Packet: Last packet sent to Mary was 100 bytes, packet succeeded: true
Total bytes now sent to Mary is 400
Client Mary disconnected
```

Clients: David, Peter, Matthew, Mary

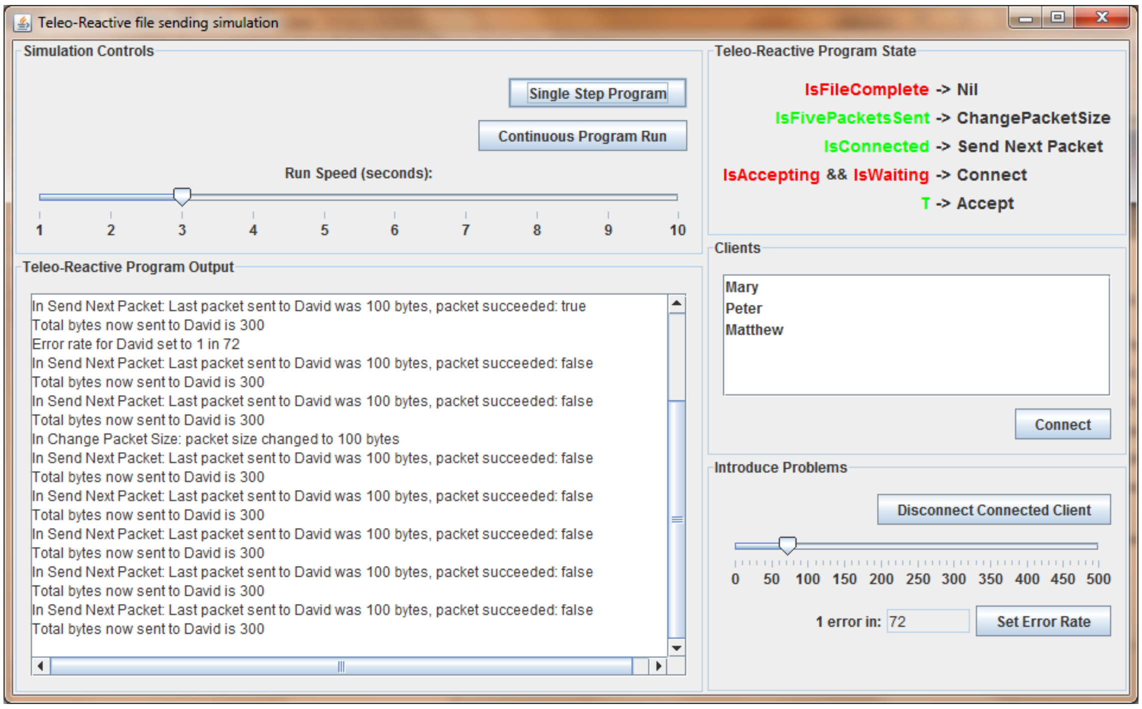
1 error in: no errors

Figure 5.3: A client has become disconnected from the file sending service

Once the new error rate has been set and a multiple of five packets has been sent to the client, 'IsFivePacketsSent' becomes true and this rule now takes priority over 'IsConnected' so 'ChangePacketSize' executes. None of the previous five packets successfully sent so the packet size is reduced if possible to accommodate. 'ChangePacketSize' actually calls the 'SendNextPacket' action once at the end of execution so that 'IsFivePacketsSent' becomes false. Without this 'IsFivePacketsSent' would always be true and any rule with a lower priory could never fire. The same procedure is true when the error rate is decreased, i.e. the packet size increases with a better quality connection, as shown in figure 5.5.

5.1.3 Conclusions

The file-sending example provides a demonstration of how T-R might be used in high-level tasks and the benefits of this approach should be made apparent. Essentially, a T-R program works towards its goal with the opportunity to re-arrange the process of achieving the goal. Rules are not triggered in procedural way but are



```
IsFileComplete -> Nil
IsFivePacketsSent -> ChangePacketSize
IsConnected -> Send Next Packet
IsAccepting && IsWaiting -> Connect
T -> Accept
```

```
In Send Next Packet: Last packet sent to David was 100 bytes, packet succeeded: true
Total bytes now sent to David is 300
Error rate for David set to 1 in 72
In Send Next Packet: Last packet sent to David was 100 bytes, packet succeeded: false
Total bytes now sent to David is 300
In Send Next Packet: Last packet sent to David was 100 bytes, packet succeeded: false
Total bytes now sent to David is 300
In Change Packet Size: packet size changed to 100 bytes
In Send Next Packet: Last packet sent to David was 100 bytes, packet succeeded: false
Total bytes now sent to David is 300
In Send Next Packet: Last packet sent to David was 100 bytes, packet succeeded: false
Total bytes now sent to David is 300
In Send Next Packet: Last packet sent to David was 100 bytes, packet succeeded: false
Total bytes now sent to David is 300
In Send Next Packet: Last packet sent to David was 100 bytes, packet succeeded: false
Total bytes now sent to David is 300
In Send Next Packet: Last packet sent to David was 100 bytes, packet succeeded: false
Total bytes now sent to David is 300
```

Mary
Peter
Matthew

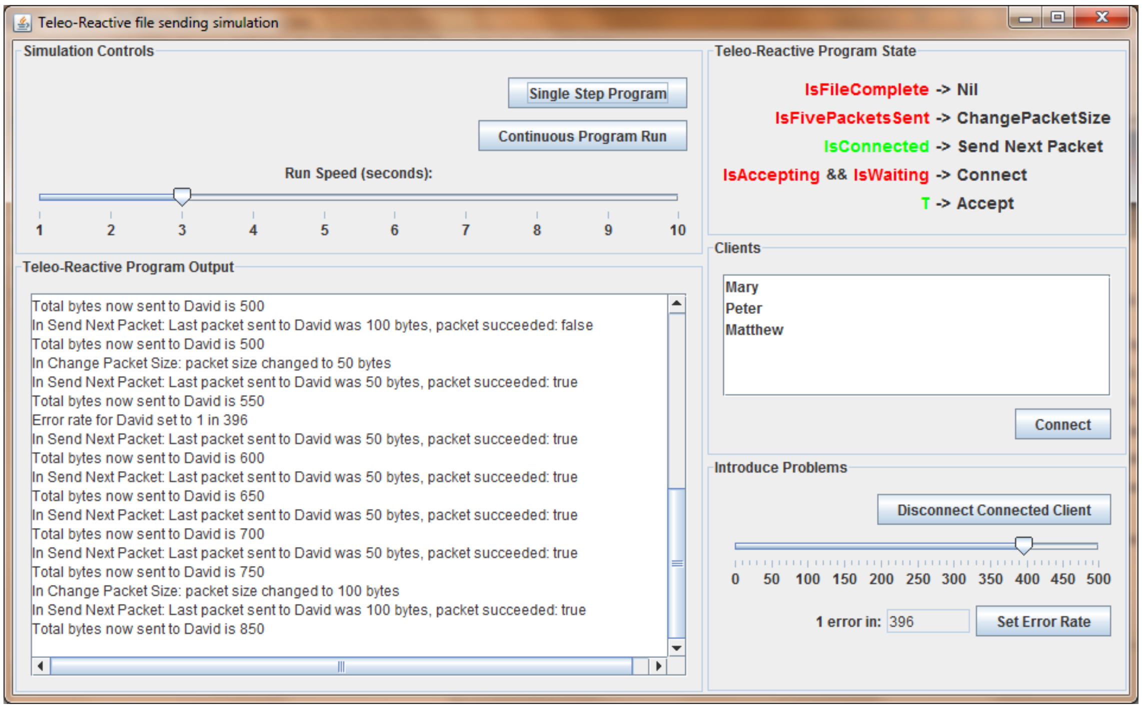
1 error in: 72

Figure 5.4: File-sending example (Self-optimising behaviour shown)

continually evaluated so the program can enter the appropriate rule at any point, be it rolling back to a previous rule or perhaps skipping several rules if circumstances dictate it.

The self-healing nature of T-R programs should be apparent as faults injected into the running program did not cause an issue, instead the system took avoiding action and continued to operate correctly. This robustness is apparent in all T-R programs and it could be said that self-healing is an integral and natural part of all T-R programs depending on the interpretation of self-healing.

The self-optimisation where packet sizes are changed in accordance with runtime communication quality is specific to this file sending example and not inherent to T-R programs themselves as with self-healing. However it was simple to add this characteristic (being a single and simple rule) and the T-R paradigm naturally presented itself to this. It could also be argued that the program is self-configuring as the T-R program adapts to the currently connected client. In fact any self-* property could be easily woven into a T-R program and doesn't have to be added as part of the initial program designs either.



```
Total bytes now sent to David is 500
In Send Next Packet: Last packet sent to David was 100 bytes, packet succeeded: false
Total bytes now sent to David is 500
In Change Packet Size: packet size changed to 50 bytes
In Send Next Packet: Last packet sent to David was 50 bytes, packet succeeded: true
Total bytes now sent to David is 550
Error rate for David set to 1 in 396
In Send Next Packet: Last packet sent to David was 50 bytes, packet succeeded: true
Total bytes now sent to David is 600
In Send Next Packet: Last packet sent to David was 50 bytes, packet succeeded: true
Total bytes now sent to David is 650
In Send Next Packet: Last packet sent to David was 50 bytes, packet succeeded: true
Total bytes now sent to David is 700
In Send Next Packet: Last packet sent to David was 50 bytes, packet succeeded: true
Total bytes now sent to David is 750
In Change Packet Size: packet size changed to 100 bytes
In Send Next Packet: Last packet sent to David was 100 bytes, packet succeeded: true
Total bytes now sent to David is 850
```

Figure 5.5: File-sending example (Packet size is increased with a higher quality connection)

5.2 Datacentre

This section asks whether the T-R paradigm can be effective as a manager for a datacentre environment. A simulation of a datacentre is set up which also shows the use of the temporal analysis feature of JTRAF as a visual feedback tool. We show how the data shown in TAP is automatically logged to a file during the program run. The logged data can be used to verify expected results against actual results, thus aiding debugging and verification. We can also generate graphs from the data and make improvements to the system based on our analysis.

5.2.1 Condition definitions and manager setup

The sections of the datacentre include the model which is used to describe the current state of the simulation and does not contain any behaviour itself, the T-R manager for control of the model according to perceived environmental events, and an automated event generator. All these elements are described in section 4.2.

These experiments will be using two xml scripts as sources of requests for our datacentre simulation ‘requests.xml’ and ‘trendingdata.xml’ as listed in appendix A where ‘requests.xml’ generates input of a steady but fluctuating nature and will be active for approximately thirty minutes. ‘trendingdata.xml’ will continue to generate requests until the system is stressed (the capacity for requests is less than the current ammount of requests) which will steadily drop to a managable level. This script will be active for approximately one hour. In each of the graphs, the vertical dotted line represents the approximate end of the script input. After this point the request level will remain at the level attained at the end of the script.

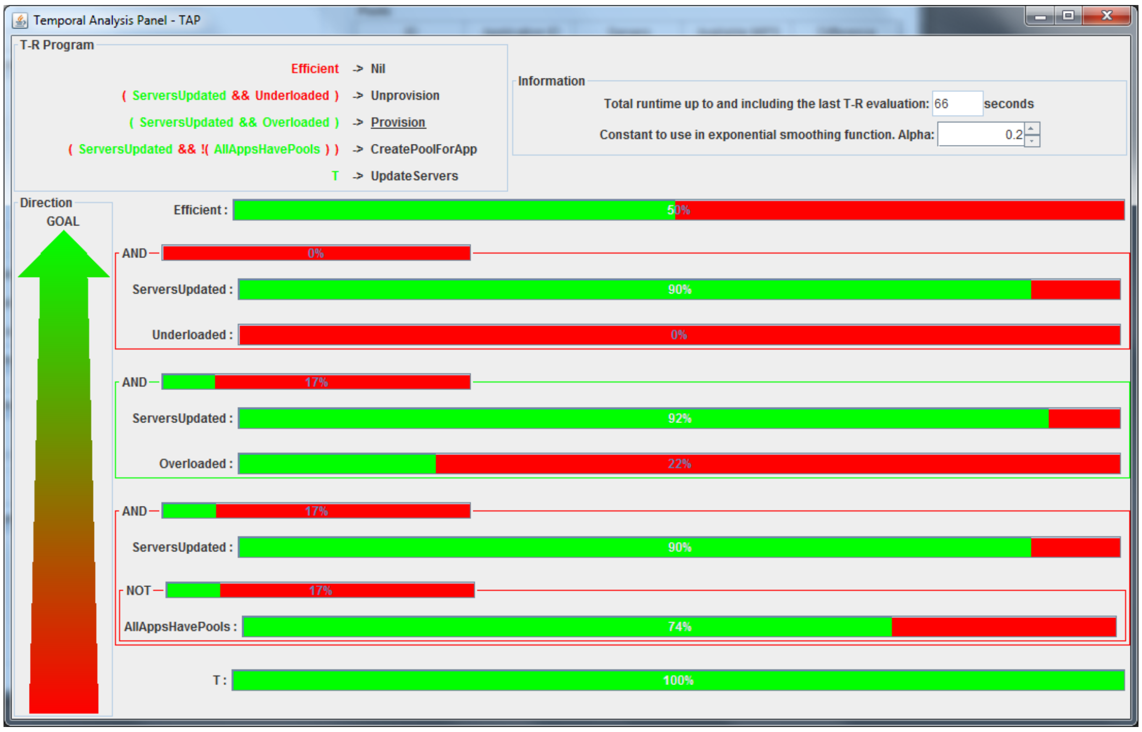
In each of the scripts we generate incoming requests for four applications every six seconds. This provides an adequate level of complexity to be managed but these may or may not be realistic values. The purpose of the experiments is to show the use of T-R in situations similar to a datacentre scenario and not to realistically represent a datacentre.

There are also three T-R managers available which can be used in the simulation run. These are a standard manager which uses a single T-R manager to distribute servers to the appropriate pool and increase its capacity. The advanced T-R manager has the same aim but uses a sub-T-R program to distribute servers more evenly. The third manager also has the same aim but uses exponential smoothing to predict the state of the model in the future and provision servers accordingly. All three have the same main goal of ‘Efficient’ so this is used in conjunction with the TAP logging system to compare the three managers.

The ‘Efficient’ condition is true when each pool in the datacentre is not ‘Overloaded’ and not ‘Underloaded’. ‘Overloaded’ can be defined as at least one pool does not have enough capacity to cope with the current incoming requests. ‘Underloaded’ can be defined as at least one pool has more capacity than is necessary for the current request level and at least one server can be returned to an idle state without causing an ‘Overloaded’ condition.

5.2.2 Self-Configuring Behaviour

The datacentre is required to configure itself repeatedly and reliably to satisfy the incoming requests. The T-R managers all configure and re-configure the datacentre model to varying degrees by moving servers as the current state requires. Lets say for example that there are four applications present in the datacentre and applications 2 and 3 receive some incoming requests. When any of the managers are running the first and immediate change is that the 'overloaded' condition changes from false to true as now there are more requests than the datacentre can provide for, as shown in figure 5.6. What is needed is for servers to be provisioned to satisfy the requests. The T-R manager provisions servers until the 'overloaded' condition becomes false again.


Temporal Analysis Panel - TAP

T-R Program

```
                                     Efficient -> Nil
           ( ServersUpdated && Underloaded ) -> Unprovision
             ( ServersUpdated && Overloaded ) -> Provision
( ServersUpdated && !( AllAppsHavePools ) ) -> CreatePoolForApp
                                             T -> UpdateServers
```

Information

Total runtime up to and including the last T-R evaluation: 66 seconds

Constant to use in exponential smoothing function. Alpha: 0.2

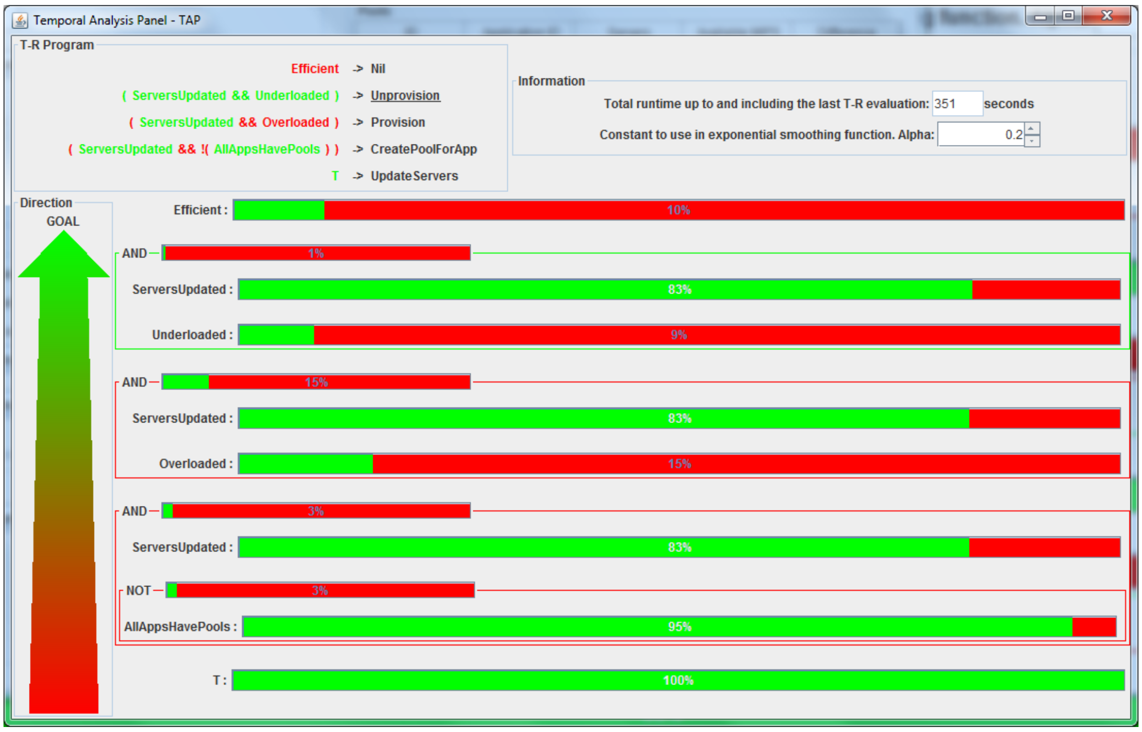
Direction

GOAL

```
Efficient : 50%
AND— 0%
    ServersUpdated : 90%
    Underloaded : 0%
AND— 17%
    ServersUpdated : 92%
    Overloaded : 22%
AND— 17%
    ServersUpdated : 90%
    NOT— 17%
        AllAppsHavePools : 74%
T : 100%
```

Figure 5.6: TAP display of a new overloaded condition state

We also need servers to return to an idle state when not needed so that other applications can take advantage when they become overloaded. Lets say the request for application 3 has been reduced significantly. As the overloaded condition became true when new requests arrived, the underloaded condition immediately becomes true when the request level decreases for application 3, as shown in figure 5.7. Some of the servers which were previously provisioned for application 3 are unused and the T-R manager 'unprovisions' them, hence the T-R manager re-configures the datacentre.


Temporal Analysis Panel - TAP

T-R Program

```
Efficient -> Nil
( ServersUpdated && Underloaded ) -> Unprovision
( ServersUpdated && Overloaded ) -> Provision
( ServersUpdated && !( AllAppsHavePools ) ) -> CreatePoolForApp
T -> UpdateServers
```

Information

Total runtime up to and including the last T-R evaluation: 351 seconds

Constant to use in exponential smoothing function. Alpha: 0.2

Figure 5.7: TAP display of a new underloaded condition state

In this scenario, the datacentre is configured, i.e. servers are moved to their appropriate destinations to satisfy the new change in requirements. The datacentre model never does any configuring itself (the datacentre model is just a location to store information and hardware) it is the T-R manager which configures the model, so if the T-R manager is written correctly then we could consider it self-configuring. We could also regard the T-R manager itself to be self-configuring as it alters its own configuration (currently running actions change and conditions alter state) as environmental conditions change.

5.2.3 Comparing the managers: Efficient condition

Logging the efficient condition is an effective comparison for the T-R managers because it is the same condition and has the same meaning for each of the managers and therefore the results are based on the same concept, though the data should differ. It is also the goal condition, so by using it as a comparison we are effectively saying 'for how long has the datacentre been performing at its peak?'. We could then use the logs to develop more efficient T-R managers for our datacentre.

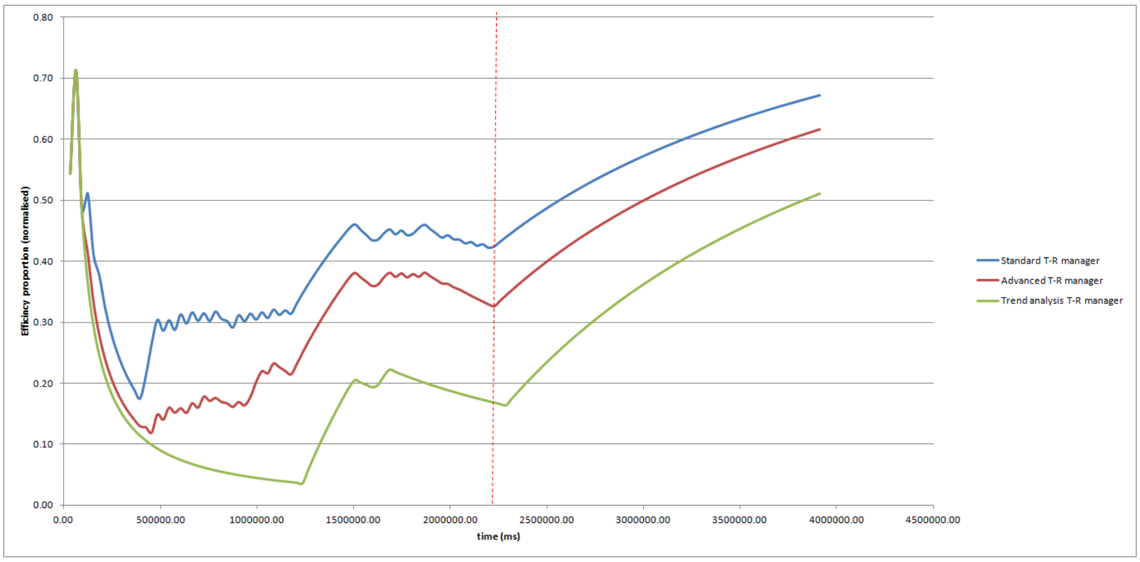


Figure 5.8: Graphical comparison of the efficient condition between the 3 T-R managers

Figure 5.8 graphically compares the three managers performance using the 'requests.xml' script. The 'y' axis shows the normalised proportion of time the program has been 'Efficient' where 'Efficient' is the program goal and the 'x' axis shows the time from the start of the program run. Therefore, where the line is higher, the longer the program has spent in the 'goal' state. The dotted vertical line shows the point where the script file ended. After this point the managers continue to operate on the final request values that was set by the script. As shown, the standard and advanced managers were most efficient with the trend analysis manager performing relatively poorly. The fluctuations in the curves are a result of the managers response to the request changes which arrive from the script.

The advanced manager is similar to the standard one and only differs in action type. In the standard manager the ‘Provision’ action is a single action type and simply provisions an available server to the next available overloaded pool. Whereas in the advanced manager ‘Provision’ is a sub-T-R manager itself and tries to provision servers to the most overloaded pool first. Even taking a server from one pool to give to another if no idle server exists.

Whilst this does not have much impact on efficiency it does ensure that servers are distributed fairly. Section 4.2.5 and figure 4.13 describe and show this. Since there is less processing in the standard manager, we would expect it to be quicker at decision making and therefore should be slightly more ‘Efficient’. Figure 5.8 reflects this.

The curve for the trend analysis manager shows it is far less efficient than the other two managers. As the script contains less of a trend in generated requests, i.e. the data generated by the script does not follow any specific pattern, this lack of efficiency is to be expected. The trend analysis manager tries to compensate for a trend which does not exist.

5.2.4 Comparing the managers: Overloaded condition

Although ‘Efficient’ is a good condition to use for comparison, sometimes we want to evaluate another condition. We know ‘Efficient’ is true when the system is neither ‘Overloaded’ or ‘Underloaded’ so it could be that problems can be tackled by narrowing our scope and check just the ‘Overloaded’ condition. We might find that although the standard manager is more ‘Efficient’, another manager is better at dealing with an ‘Overloaded’ state.

Figure 5.9 shows the same script being processed but with the ‘Overloaded’ condition being graphed. The ‘y’ axis is the normalised proportion of time that the datacentre has been in the overloaded state. The ‘x’ axis is the time since the start of this simulation instance. We see a large increase in the overloaded state

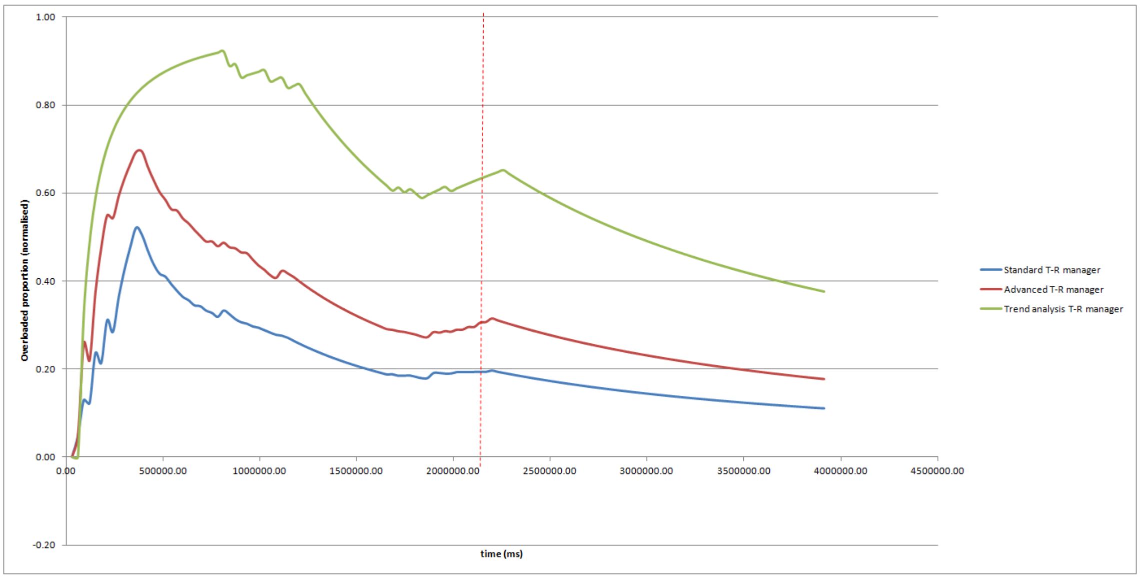


Figure 5.9: Graphical comparison of the overloaded condition between the 3 T-R managers

at the start of the program run as the manager responds to the new request data, with the curve showing a decrease in the overloaded state as time increases. We would expect the datacentre to be overloaded towards the beginning because each of the applications contain zero servers initially and therefore there would be a huge difference between the number of incoming requests and the servers provisioned to cope with these requests. It might be worth persisting the information about the average number of servers needed so that the managers can better compensate in the initial stages. It is worth noting that for any T-R program, all the temporal data which makes up the conditions within the program are automatically logged. This is done by JTRAF, so there is no need for the developer to perform any logging actions themselves.

In the case of figure 5.9, the higher the line, the more longer the system has been overloaded, so in fact a lower line is better in terms of being overloaded whereas the opposite is true for the graphs of the ‘Efficient’ condition. This is consistent with figure 5.8 and shows that the standard T-R manager is better and that being ‘Overloaded’ is likely a major factor in this.

5.2.5 Varying the server provisioning time

To show how the logging mechanism can be used to compare one T-R program with another, perhaps to locate potential for improvements or to verify the newly developed T-R system is performing as it is expected, we compare the 'Efficient' condition of the three managers with respect to the server provisioning time. Figures 5.10 to 5.13 show the graphed results of the experiments where server provision time is increased from 30 seconds in the first experiment to 240 seconds in the last experiment. Each experiment was run for one hour using the 'trendingdata.xml' script.

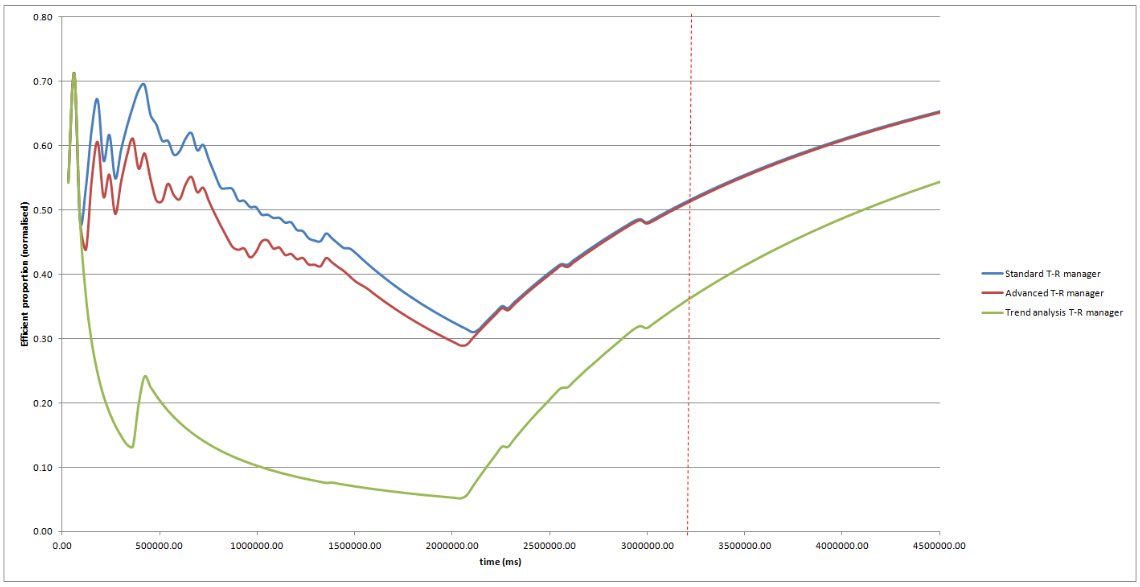


Figure 5.10: Graphical comparison of the 3 T-R datacentre managers where all servers have a 30 second provision time

We had expected the trend analysis version of the datacentre manager to perform better than the other two managers but as we can see from the graphs, this is only the case when the server provision time is above four minutes and only when the pools are overloaded. Around the 2100000 millisecond mark the script begins to reduce the incoming requests for pools and thus the pools become underloaded and the standard and advanced managers show an improvement over the trend analysis manager on overall efficiency. The graph of 5.13 shows the lines from the standard and advanced managers completely obscuring each other. This is because of the

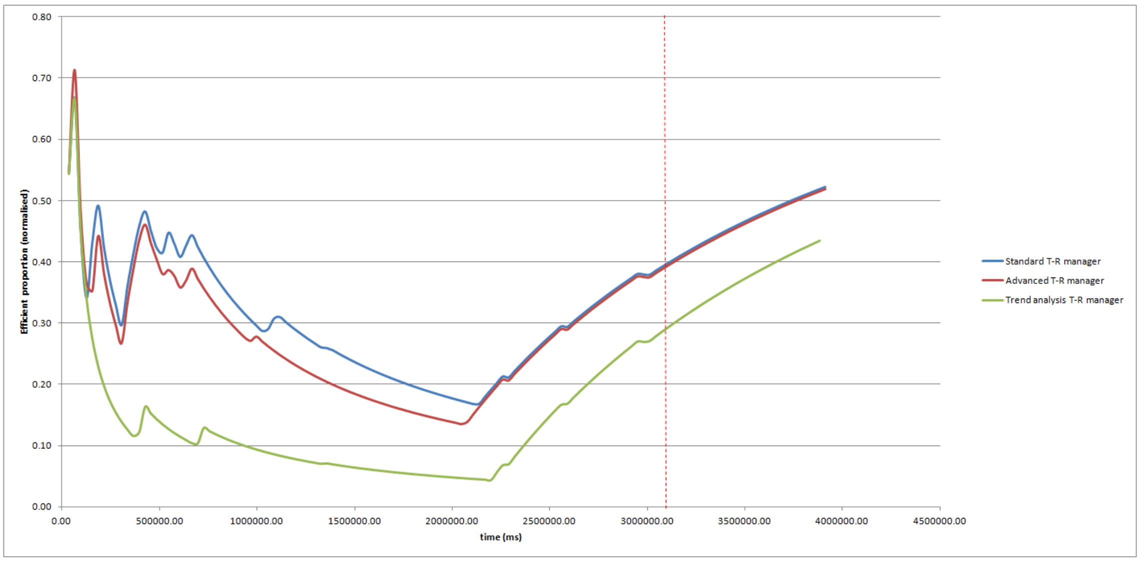


Figure 5.11: Graphical comparison of the 3 T-R datacentre managers where all servers have a 60 second provision time

similarities between these two managers, similarities which are exaggerated by the high provision time.

This is partly to be expected because a short server provision time and a relatively short trend in requests means the standard and advanced managers have adequate time to react to changes in incoming request data. Whereas a longer server provision time and a longer trend means servers need to be provisioned in advance of being needed in order to have sufficient capacity to deal with incoming requests. The trend analysis manager is less effective and even detrimental to performance after requests are reduced because here servers are returned to idle and there is no provision time associated with this action, therefore there is no need to predict a trend.

5.2.6 Varying the lookahead time for trend analysis

Given that the trend analysis manager was less 'Efficient' than the other managers in all cases apart from where the server provision time is relatively high, our next experiment varies the lookahead time for the trend analysis manager to investigate the effect on efficiency.

Figure 5.14 shows the effect of running the trend analysis manager with the

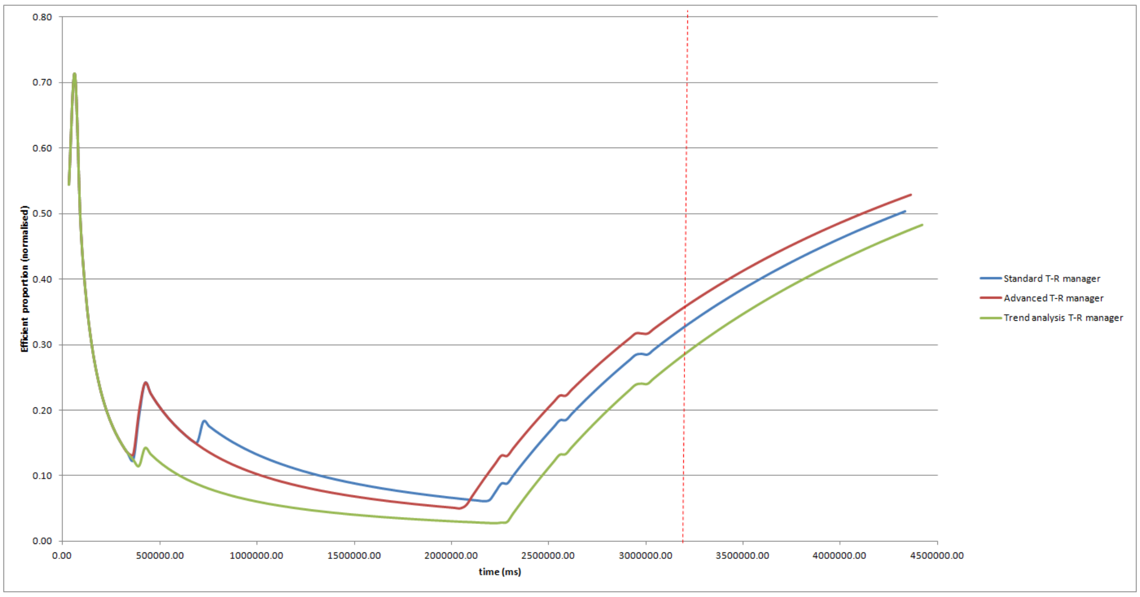


Figure 5.12: Graphical comparison of the 3 T-R datacentre managers where all servers have a 120 second provision time

lookahead time set to 5, 10 and 20 steps. The graph also shows the standard manager for comparison purposes. Each manager run uses a server provision time of 120 seconds.

From this experiment we see that varying the lookahead time had very little impact on the efficiency of the system and no variation showed any improvement over the standard manager. A change in server provision time had a far greater impact on efficiency.

5.2.7 Conclusion

The datacentre example made extensive use of the TAP extension and the associated logging mechanism to both design and analyse the T-R manager systems for the datacentre. We used the logs to compare the managers and to analyse the results of changes. Specifically we made several changes to the trend analysis manager and used the TAP logger to analyse the results.

We had expected the trend analysis T-R program to perform better than the other two programs except this was not the case. Further investigation revealed that the reason for this is likely due to the coding of actions and conditions them-

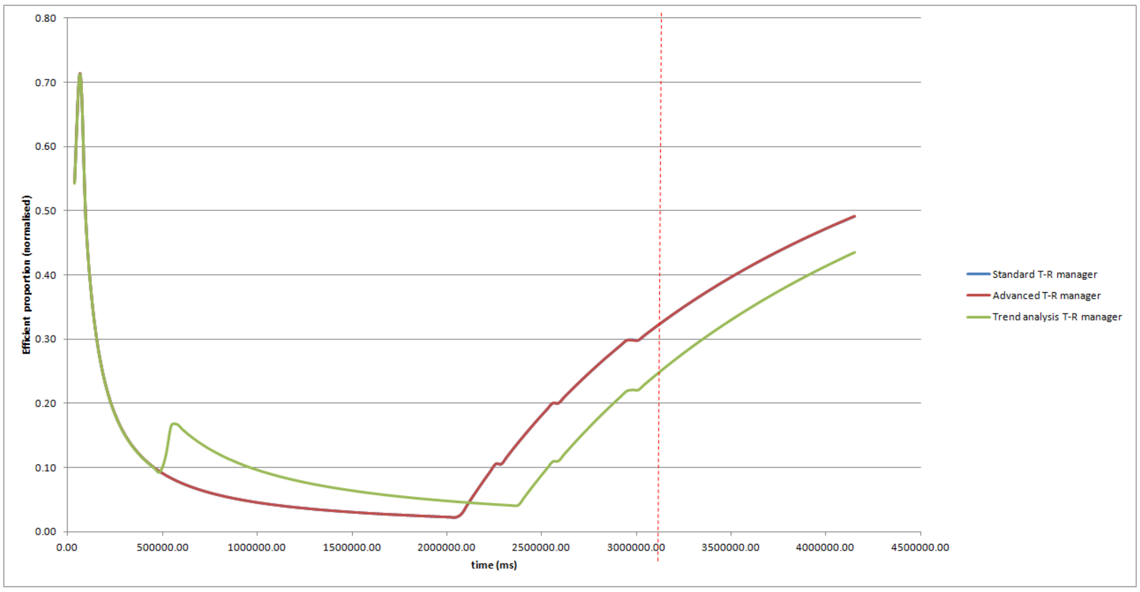


Figure 5.13: Graphical comparison of the 3 T-R datacentre managers where all servers have a 240 second provision time

selves rather than any settings or configuration of the input. Therefore, further investigation of any improved T-R manager is needed.

The benefits of TAP were evident however. It allowed easy comparison of the three managers used and allowed positive and negative effects of changes to be recorded and visualised. As TAP and its logger is built into JTRAF, the same visualisation capabilities, logging and graphing could be used on any program that is built with the JTRAF framework.

5.3 Summary

In this chapter we have presented two examples and based some of our experiments in them in order to test T-R programs as managers of their intended systems. The first example is of a simple file sending program which simply sends a single file to simulated connected clients. This example is not designed to improve on, or indeed be a particularly accurate representation of a real and similar system but is merely designed to demonstrate the use of T-R as a manager and to inspire some designs of the use of T-R in similar systems. We also used it to test robustness levels of the

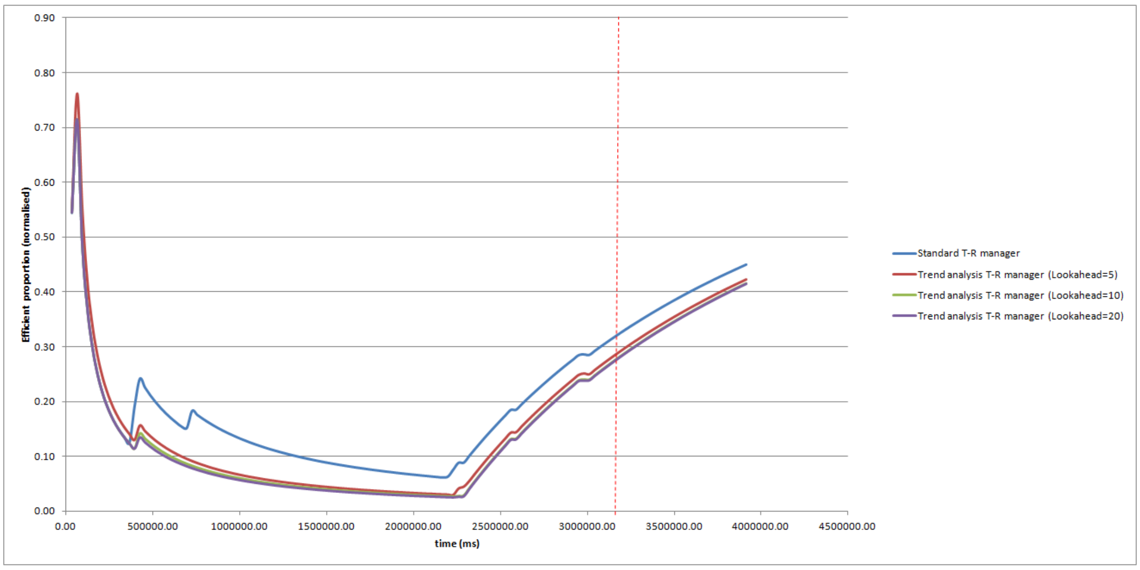


Figure 5.14: Variation of the lookahead time for the trend analysis T-R manager and its impact on efficiency

system and investigate some self-* properties which T-R programs naturally provide and can be easily adapted for.

The second example of a datacentre simulation attempts to optimise the use of components through our T-R manager in order to deal with incoming requests. There are three main parts to this example; the datacentre model, the T-R manager and the automated request generator which is configured with xml scripts. Again, none of these elements are accurate representations of real ones but is designed to inspire the use of T-R managers in more complex real datacentres. In this example we have made extensive use of the extensions built into JTRAF to initially design the managers and to compare, the three managers.

In the file sending example, faults were added, and changes were made to the running system to observe how the T-R management system coped. In the example, we were able to connect and disconnect clients at any stage in the process, as well as adjust the noise levels for the connection, up and down. Not every possible unknown fault could be injected to the system but the faults did demonstrate how the system coped with unexpected changes. Since all T-R systems are the same, i.e. they are composed of condition-action pairs (rules), any T-R system will simply move to a more appropriate rule when an unexpected event occurs in the same way that this example did, i.e. by focusing on the conditions which are true and false and adjusting the state as appropriate, and not focusing on the fault itself. The T-R system could be regarded as self-healing because when an unknown event occurs the system 'rolls back' to a state which it is capable of achieving and working towards the point of previous failure. If the system succeeds this time it will progress past this point, if not, it may need to 'roll back' again, but at no point did the system cease operating completely.

This is perhaps why software systems of the past often fail. Trying to account for every possible individual error is of course impossible because not every individual error can be accounted for. A condition such as 'isConnected' being true or false covers many possible errors, even unplanned for ones.

The use of a datacentre for the second example was chosen because of the recent activity from the autonomics community around datacentres, thus it is a well established and interesting area for investigation. It is also a complex challenging task which can prove to demonstrate the T-R mechanism qualities. Despite this complexity, we discovered how simple and concise a T-R system can make the process with a view to the management of the task. Automated and simulated requests for the datacentre were generated via a scripting process and three T-R managers were constructed to attempt to efficiently deal with the requests. The main purpose of this example was to investigate the use of the TAP mechanism for comparing, improving and constructing the three managers.

The TAP logger automatically records the true time for each condition in the program run. This recorded data was used to generate graphs. The efficient condition was first plotted on the graph for all three T-R managers using the 'requests.xml' script to generate the same simulated data for each manager. The expected result was that the trend analysis managers would show an improvement over the other managers because it is able to pre-empt changes by analysing the trend and provisioning servers in advance. However the graph of figure 5.8 shows that the opposite is true. i.e. the trend analysis manager performs worse in terms of efficiency. Initially, it was suspected that one of the many parameters should be altered to produce a more satisfactory result. It could also be that the script used produced too random request data, i.e. there is no trend to follow. The 'trendingdata.xml' script was created to generate requests which more closely follow a specific trend.

To show that the logger can record data for more than just the goal condition, the 'Overloaded' data was also graphed using the same 'request.xml' script. Since 'Efficient' effectively says this condition is only true if the datacentre is not overloaded and not underloaded then we can get a more accurate view of the datacentre management process by inspecting the logs for the 'Overloaded' or 'Underloaded' conditions individually. Figure 5.9 shows the graph of the 'Overloaded' condition. From this we can see that the trend analysis manager is overloaded for a longer period of time than with the other two managers meaning that the time spent underloaded will likely show a negligible difference between the three lines compared with the overloaded data. This would be as predicted since the provision time for each server only has an effect while the pool does not have enough servers and therefore only 'Overloaded' would be affected. The time taken for a server to return to idle is always 0.

Using the new trendingdata.xml script to generate more predictable request volumes the server provision time is varied and the graphs are shown in figures 5.10-5.13. This experiment was designed to investigate the impact on efficiency of varying the server provision time for each of the three managers. We had expected the trend analysis manager to show an improvement over the other managers as the server provision time increased coupled with the trendingdata.xml script, which should be optimised for the trend manager. The results confirm this prediction as the efficiency levels of all the managers became closer as the server provision time increased. However, the level of efficiency remained very low for the trend manager in each test. The reason for this poor performance of the trend manager could be that another variable might need changing to improve efficiency or perhaps the script and the produced trend data simply do not run for an adequate length of time to show a greater improvement of efficiency in the trend manager.

The next test involved altering the lookahead time for the trend analysis manager. This is the number of stages in the future to attempt to predict a trend (smaller values should yield more accurate predictions but less able to predict a trend beyond a few steps). The lookahead time was predicted to be a likely cause of the poor performance of the trend manager so testing different values seem like a valuable approach.

As can be seen from figure 5.14 adjusting this value between five and twenty did not have much of an impact on the outcome. In fact, a lower value showed a slightly higher efficiency whereas it was expected that a higher value would. The standard manager was still more efficient than any other manager.

These results, whilst surprising are not problematic to the project aims of demonstrating the use of T-R in autonomic systems. and demonstrating the importance of the extensions for design and development.

5.3.1 Contextualising the research questions

The file sending example is designed firstly to test the use of T-R as a simple manager of autonomic systems and also as a showcase for the benefits of T-R in this and other similar systems. The file sender example demonstrated the robustness of a T-R manager by injecting faults to the running system, showing how the T-R system coped with them. The viewer should be brought to an understanding of how T-R could cope with external influences in other systems. It also showed the simplicity of building autonomic self-* properties into T-R.

In the datacentre example, we showed how the use of sub-T-R programs could help create a very extensible system whilst keeping the simplicity and re-usability of a T-R program. In this we answer the question of “To what extent can Teleo-Reactive programming techniques (T-R) be used to construct autonomic management systems and what are the benefits of doing so?"

In chapter 4 the construction of a simple T-R program was shown. In this and in Hawthorne and Anthony [102] the difficulties are highlighted and the reader should appreciate that the challenges of developing a T-R program are unique when compared with a program written in another more standard language. That said, a T-R program, once developed has proven to be very robust in its mechanism for coping with expected and unexpected events.

The development issues can be addressed with a good method, as explained in [102] and we have also aimed to reduce the issues with built in mechanisms to JTRAF. TAP provides both qualitative and quantitative feedback in the form of a visual state representation and a logging mechanism for this state information. Also, because TAP is built into JTRAF, any future work to reduce these issues further can be built into the same framework so that all T-R programs using JTRAF will get the same benefit. Some of these future work plans are explained in section 6.4.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 On Autonomics

When the term ‘autonomic computing’ was first introduced by Kephart and Chess [1], the term was never ‘well defined’. This was good because it meant researchers were left a lot of scope for potential ideas and developments into this domain. Now it seems that the original ideas are still as abstract as they were when they were first introduced and without a solid and indisputable definition it is difficult to progress the field in a positive direction. This is evidenced in the fact that the majority of autonomic systems are designed in complete isolation to other systems and concentrate on a solution to one specific problem rather than developing a generic autonomic tool. Quite often these solutions are developed from scratch as shown in [110, 50, 111]. These incorporate autonomic principles but do not build on pre-existing autonomic system designs which leads to an ever expanding variety of ideas without forward progression.

During this project, the lack of a definition has made our own work difficult to quantify because there is no accepted method of measure or scale for autonomic systems. Therefore, claims that a system is self-optimising for example can only be based on a dated and ambiguous definition of self-optimisation. There is the autonomic maturity index [67] written by IBM but this description remains unchanged and unchallenged since its introduction in 2003. Furthermore it was assumed that most software resided at the lower levels on this scale but many software products actually occupy higher levels of autonomicity.

A method of measuring autonomic solutions would be hugely beneficial because new autonomic ideas and architectures etc. could be substantiated as a quantity rather than an abstract quality and then possibly improved. Autonomic software measurement is a problem that has been discussed in Huebscher and McCann [65] where metrics are proposed as a possible measurement solution. These are good suggestions but gaining widespread acceptance is difficult because there are a general lack of rigid definitions and no standards in the first place.

When attempting to answer the question ‘Can T-R be used in an autonomic system?’ it is difficult to provide a definitive answer but this has not been a barrier to the project because autonomic software does address the very real problem of software complexity and as we have shown, T-R programs can reduce this problem significantly whilst fitting the description of some self-* properties.

The central idea behind autonomics seems an ideal solution to software complexity, i.e. to handle some tasks in the background by the system itself and thereby reducing the cost of ownership to a manageable level. This is a sound approach because we are acknowledging the fact that complexity in software systems will always exist and instead of attempting to discover a method to remove complexity entirely, autonomic systems are essentially attempting to automatically deal with some of the complexity so that the remainder is manageable.

From a psychological perspective, if an autonomic system removes human influence from a system, is it safe to do so (see Trust, Validation and Verification in section 2.2.2) and by claiming a system is autonomous in some way, does this not relax the system operator to the extent where they neither have the skills nor the readiness to deal with situations when they do occur because of the over-reliance on the ‘automated’ system? Furthermore, would a human operator have the skills to alter the system to avoid the problem re-occurring? A problem which before the system was ‘automated’ would be easily avoided by manual checks and inspection. The human operator would also be alert to this and other unexpected events, putting them in a much better position to fix or even avoid the problem altogether.

Automation has led to far fewer accidents where human lives are concerned as Robert N. Charette explains in this IEEE Spectrum online article [112]. The problem is that an over-reliance on a previously flawless system means that the rare incidents are not prepared for by the operator. Charette gives several examples of these incidents, including the case in June 1995 where the cruise ship Royal Majesty ran aground. Here, the cable for the GPS antennae had become detached from the antennae and had put the system into a degraded mode where sea changes and wind speed were not taken into account consequently the autopilot system which the ship was operating on had put the ship off course. Before ‘automation’ the crew would likely be manually monitoring the horizon, plotting course and making adjustments to speed and direction.

The error was almost impossible to locate, and even if it had been found, the operator would probably not be able to directly fix it. As the system becomes more reliable, the operators become more complacent in trusting its abilities.

The incidents concur with the findings of psychologist Lisanne Bainbridge, who identified several ironies and paradoxes of automation [113]. The irony, she said, is that “the more advanced the automated system, the more crucial the contribution of the human operator becomes to the successful operation of the system.” Bainbridge also discusses the paradoxes of automation, the main one being that the more reliable the automation, the less the human operator may be able to contribute to that success. So in fact, if the system designer heavily automates the system, they are in fact locking the human operator out of the process, i.e. making it harder for the human operator to intervene. The danger here is that the designers themselves assume the system is faultless and will never need fixing or maintenance, which of course is never the case. Perhaps the human element should be more tightly integrated into the ‘automated’ system rather than attempting to eliminate it altogether.

6.2 On the use of T-R in Autonomics

Autonomic systems in use today are often tailored towards the specific problem which they are trying to solve, meaning the autonomic system may not be usable in other projects. This means that autonomic research does not build in a progressive way as each new system is developed (expanding outwards rather than progressing forwards). The autonomic problem is often viewed as an architectural challenge, or one for a policy based system, or one for utility functions. In fact there have been several approaches to the autonomic problem but this project has been investigating the use of a T-R approach.

T-R programs are a unique approach to the problem in that it is a logical programming paradigm and a general approach to the autonomic challenge. With T-R management systems we can treat problems in much the same way as a human brain considers and tackles a task. That is in a logical and procedural way but is always capable of responding to external and unexpected events. The procedure can then be rearranged once or many times to compensate. This detection of events and state and rearrangement of tasks is an inherent part of the T-R system, therefore reducing the need for external maintenance and puts T-R in a position to be a genuine alternative to existing autonomic approaches.

Much of the work in this project is based on the T-R paradigm which was never originally heralded as an autonomic system but an autonomous system for controlling low-level agents and robots. This leads to the question of how to differentiate between autonomic and autonomous systems. The initial thought might be that autonomic systems are composed of self-* properties where autonomous systems are not. However, these too are poorly defined and it could be claimed that many autonomous systems could in part be re-introduced as autonomic systems without any changes. This is true of T-R as well. We claim that T-R programs can be self-healing because they can recover from failures and continue to operate without any external intervention. Whilst self-healing behaviour is an inherent part of T-R, it may be possible to introduce other self-properties as shown in the file sending example where self-optimisation was included with the addition of a single rule.

Using a T-R approach reduces the maintenance cost because the system uses true of false conditions to define the current state. These conditions can be changed by run-time errors, some of which were not expected at design time, and the T-R program takes appropriate steps to define a more appropriate state. Therefore, the conditions encompass many errors and it may not be necessary to handle them specifically. Compared to a non-T-R system the errors would likely be required to be fixed by a human immediately to prevent any further system fatal errors. This does not prevent other unexpected and as yet undiscovered errors. So constant maintenance is highly likely.

Using T-R as an approach to building autonomic systems has many benefits, including its simplicity in writing a program and its simplicity in structure compared to other approaches which often use many complex components to form a solution. This approach is further simplified by the developed framework (JTRAF) which provides a T-R interpreter and extensions to aid development and maintenance as well as consolidating much of the system complexity within. T-R programs are very robust and are able to recover from many unexpected errors. This, it could be argued, is a form of self-healing, where other self-* properties are easily incorporated into a program as shown in the file sending example.

One of the challenges in designing JTRAF was that the type of actions normally performed for low-level agents are not the same type of actions in a high-level application. T-R was designed for low-level agents and as such it is very effective in this area. In a high-level program, many of the actions could be atomic so interrupting the process here (as a traditional T-R program does) could lead to disastrous consequences. In order to adapt T-R for these high-level processes, we have developed

JTRAF to allow an action to run to the end of its cycle should the corresponding condition not hold true. Therefore the action is allowed to gracefully exit rather than being forced to interrupt a perhaps atomic operation.

6.3 Contributions

This project has raised some interesting questions about autonomic computing in general and has challenged the ideas and concepts surrounding this domain. It has even raised some questions about the methods used in some computer programming projects. The idea that we can catch all problems that will occur during a software life-cycle and fix the rest when they first arise is a naive thought and should be rejected in all but the most trivial programs, for the simple reason that software can never be error free, there will always be several errors which the developer has not provisioned for. Instead we should move to a goal-oriented approach where known and unknown events are detected through the state of the program that those events cause. The program should focus on the final goal and sub-goals, which will drive it in a positive direction at the same time as casting a wider net on known and unknown events. The goals and sub-goals are far fewer than the number of potential errors and therefore a software product which focuses on the former rather than the later produces more robust and manageable software.

The project has identified some issues with the original concepts of autonomic computing. Concepts which, although vague, have never been adequately challenged. For example, the original autonomic computing paper described the self-* properties, and their meaning. However, as we have revealed through our T-R examples, system designers could easily argue the case that they contain these properties despite not having been designed to and without valid argument to the contrary. In fact systems which were originally designed as autonomous could be re-introduced as autonomic without many or any changes. The project has therefore been thought provoking and encourages an alternative view on this domain.

This project has provided a genuine alternative to other methods employed as autonomic frameworks. The T-R system is unique because it offers a simple to use and understand method which is not present in many other systems. Although the method is simple it offers developers the chance to extend their programs to include properties which are highly desirable in today's software. T-R programs are also very robust meaning maintenance is reduced for the developers. Autonomic software was originally heralded as a way to reduce complexity and the large cost to maintenance that this brings. In this way, T-R programs are an ideal structure on which to base an autonomic system.

The datacentre environment is a heavily explored and current use-case among today's autonomic researchers. By implementing a datacentre example in this project we show how T-R can compete with other management systems of this type. It also shows how T-R programs can excel in this challenging environment. We have shown how a T-R system can be extended and structured through the use of sub-T-R programs in order to divide and make the problem manageable.

The ease of designing and developing a T-R program is enhanced by JTRAF, a framework which was built for this purpose. JTRAF allows T-R developers to develop their system without the need to develop or be concerned with the T-R interpreter for every project. JTRAF has been designed to allow changes and improvements to the framework without the need to change the interfaces. This essentially means that developers can integrate a new, improved version of JTRAF without this change causing any negative effects on the current software project with which it is integrated. Whilst JTRAF handles the mechanism of T-R, it also aids development and maintenance of the system through the use of TAP which provides visual feedback of the temporal state as well as logging the same information for post execution analysis.

Aside from autonomics, the work here is of benefit to T-R developers in general. Previous work on T-R focuses largely on mechanisms for evolving and learning as the program senses the environment and gains new abilities, extending the T-R basics for this purpose. These mechanisms include genetic algorithms [95, 114, 115], neural networks [116, 96] and reinforcement learning [117, 97, 118]. However, this project has shown issues with design and implementation of T-R programs which has largely been ignored by researchers. This project has attempted to address those issues as this gives a stronger foundation for further development. For example the TAP mechanism has benefits for a wider community than autonomics with regards to the entire development process.

Considering the importance of a reliable framework on which to base autonomic programs, JTRAF has been developed using a test first approach. This, in the authors experience has been a reliable technique for producing highly robust code. In TDD we first create tests that represent the correct outcome of a particular input, then write the code to 'pass' the tests. All the tests can then be run in a instant and as often as desired. Not only does this produce reliable code at design time but when improvements are made to the code, the same tests can be executed, informing the developer if any tests have failed since the changes were made. Once all tests pass, the developer can be confident in the reliability of the code. TDD is described in chapter 2. In the software created for this project 51 tests were created for JTRAF alone, 26 for the file sender and 73 for the datacentre applications. In appendix B we show a small selection of these tests which are useful for documenting the code.

6.4 Future Work

Presenting a T-R program to an interested party is a difficult task in itself. Individual conditions and actions should be described concisely enough to fit neatly on a display surface after being logically joined with other conditions but long enough to describe the element behaviour. This often leads to developers either using a well described program or a conveniently displayed program but not both. Therefore a naming convention is required or a novel way to describe and document T-R programs needs to be developed. Consider the main T-R program for the datacentre use case:

$$\begin{aligned} Efficient &\longrightarrow Nil \\ ServersUpdated \wedge Underloaded &\longrightarrow Unprovision \\ ServersUpdated \wedge Overloaded &\longrightarrow Provision \\ ServersUpdated \wedge \neg AllAppsHavePools &\longrightarrow CreatePoolForApp \\ T &\longrightarrow UpdateServers \end{aligned}$$

The ‘Efficient’ goal condition actually means:

$$ServersUpdated \wedge \neg Underloaded \wedge \neg Overloaded \wedge AllAppsHavePools$$

yet displaying this information in one place is convoluted and inefficient, therefore ‘Efficient’ is used in its place. It would be more informative to display all this information however as it more accurately describes the goal. In order for the program to correctly achieve its goal whilst not preventing other rules from executing, all the replaced conditions must be true. For example, if ‘Efficient’ no longer included the ‘ServersUpdated’ condition then the program could never run the ‘UpdateServers’ action after provisioning servers until ‘Efficient’ became false. The ‘Efficient’ goal takes priority over all other rules and ‘UpdateServers’ would be prevented from executing.

Logically joining four or more conditions in this way leads to inconveniently lengthy rules and so these conditions were embedded into a single condition called ‘Efficient’. This is however, not an ideal solution because ‘Efficient’ is far less descriptive and requires an explanation as to its purpose and contents. What is therefore needed is a method of describing a T-R program which does not require any compromise on descriptiveness and is convenient for design purposes. Perhaps a newer design model could support forward and reverse engineering.

Whilst a T-R program aids in automating the run-time processes, the composition of a T-R program often lends itself to many types of logical errors, so the composition process could be automated itself. Whilst TAP has helped this process, it could be automated further, similar to the ideas described in Hawthorne et al. [107]. In this paper we describe how actions and conditions are linked by the contributions of actions to conditions and of conditions needed for actions to execute. This effectively forms a chain of events leading to the goal. Currently, these links are implicitly stated when the program is composed, but if the links were explicitly made then new programs could be automatically constructed from new goals. For example, if we know the chain of events leading to goal A and we insert a new goal B which we state is contributed to by action 1, and this action exists in the goal A program, then it would be possible to create the new goal B program based on this information. The more programs that exist, the higher the possibility that a new program can be constructed automatically.

This automatic composition would require the developer to describe the links, at least initially. In later versions of JTRAF, links could be automatically made by learning which actions change the state of which conditions, perhaps using an artificial intelligence technique such as neural networks [96] or genetic algorithms [95].

An aid the automatic composition of T-R program could be found in Tropos [79] or i* [78] where high level models are constructed and the influences between elements of the system are shown, positive and negative. These ideas could be adapted for the T-R system as a model of the system. For example, action A might have a strong positive influence on condition A but a negative influence on condition B. Action B has a weak positive influence on condition A but no negative on condition B. Therefore modelling this might infer that action A would prevent the goal from being achieved and action B is the only possibility for use in this situation whilst the goal is still achievable.

Fortunately, in the datacentre simulation experiment all three of the T-R manager versions used the same ‘Efficient’ goal. This meant we were able to compare and contrast this condition in the three versions of the manager through TAP and the logging mechanisms. If some of the versions did not contain ‘Efficient’ then there would be no way to log and compare this condition.

What is needed is a way to add a condition to a program despite not being part of the main T-R rules of the program. This condition could then be evaluated and logged along with the other main conditions. This log could then be used to compare with other programs with the same condition. In JTRAF a rule is added with the methods:

```
myTRProg.addActionToBottom(action3);
```

or...

```
myTRProg.addActionToTop(action3);
```

where myTRProg is an instance of a T-R program and action3 is an instance of an action. The conditions contained within the action are automatically added to the myTRProg program. All these conditions are then evaluated at run-time and logged through TAP. It is also possible to add conditions explicitly to the T-R program instance using the method

```
myTRProg.addCondition(condition8);
```

however, if condition8 were not part of any rule then it would never be evaluated. With minor changes to the framework, condition8 could be evaluated and logged despite not being a part of one of the rules of the program. This would be useful if the developer wishes to know for example the proportion of time connected to a network for diagnostic purposes or for some unrelated task, but the program being developed is a room cooling manager for a large corporate server room, and is therefore unlikely to contain such a condition.

There should be an autonomic framework which is open source and contributed to by many autonomic researchers in much the same way as several current open-source projects [119, 120, 121]. This way autonomic software would have a much quicker development time, errors could be fixed much sooner and autonomic research could progress in a more positive direction. Currently techniques and architectures are developed for each individual project leading to a broad range of ideas but not much progress. T-R programming could be an ideal basis for this open source framework because it provides robust behaviour without necessarily enforcing any specific technique for self-* behaviours.

BIBLIOGRAPHY

[1] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[2] N. J. Nilsson, "Teleo-reactive programs for agent control," *Journal of Artificial Intelligence Research*, vol. 1, pp. 139–158, 1994.

[3] N. J. Nilsson, "Teleo-reactive programs and the triple-tower architecture," *Electronic Transactions on Artificial Intelligence*, vol. 5, pp. 99–110, 2001.

[4] F. P. Brooks.Jr., "No silver bullet essence and accidents of software engineering," *Computer*, vol. 20, no. 4, pp. 10–19, 1987.

[5] I. Chowdhury and M. Zulkernine, "Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?" in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. New York, NY, USA: ACM, 2010, pp. 1963–1969. [Online]. Available: http://doi.acm.org/10.1145/1774088.1774504

[6] Y. Shin, A. Meneely, L. Williams, and J. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *Software Engineering, IEEE Transactions on*, vol. 37, no. 6, pp. 772 –787, nov.-dec. 2011.

[7] T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, pp. 308 – 320, dec. 1976.

[8] D. P. Darcy, C. F. Kemerer, S. A. Slaughter, and J. E. Tomayko, “The structural complexity of software: An experimental test,” *IEEE Transactions on Software Engineering*, vol. 31, no. 11, pp. 982–995, 2005.

[9] IBM, “Autonomic computing: IBM’s perspective on the state of information technology,” http: //www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf, last accessed 2012.

[10] N. J. Nilsson, “Teleo-reactive programs web site,” http://robotics.stanford.edu/users/nilsson/trweb/tr.html, Last accessed 2012.

[11] R. J. Anthony, “Policy-centric integration and dynamic composition of autonomic computing techniques,” in *ICAC ’07: Proceedings of the Fourth International Conference on Autonomic Computing.* Washington, DC, USA: IEEE Computer Society, 2007, p. 2.

[12] P. Ward, M. Pelc, J. Hawthorne, and R. Anthony, “Embedding dynamic behaviour into a self-configuring software system,” in *ATC ’08: Proceedings of the 5th international conference on Autonomic and Trusted Computing.* Berlin, Heidelberg: Springer-Verlag, 2008, pp. 373–387.

[13] R. Anthony, M. Pelc, P. Ward, J. Hawthorne, and K. Pulnah, “A run-time configurable software architecture for self-managing systems,” in *International Conference on Autonomic Computing.* Los Alamitos, CA, USA: IEEE Computer Society, June 2008, pp. 207–208.

[14] R. Anthony, M. Pelc, P. Ward, and J. Hawthorne, “Flexible and robust run-time configuration for self-managing systems,” in *Self-Adaptive and Self-Organizing Systems, 2008. SASO ’08. Second IEEE International Conference*, 2008, pp. 491–492.

[15] R. Anthony, P. Ward, D. Chen, A. Rettberg, J. Hawthorne, M. Pelc, and M. Törngren, "A middleware approach to dynamically configurable automotive embedded systems," in *ISVCS*, 2008.

[16] M. Pelc and R. Anthony, "Towards policy-based self-configuration of embedded systems," in *SIWN System and Information Sciences Notes*, vol. 2(1), 2007.

[17] J. Hawthorne and R. Anthony, "A reconfigurable component model using reflection," in *SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems.* Newcastle: ACM, November 2008, pp. 95–100.

[18] R. Pietrantuono, "Component airbag: a novel approach to develop dependable component-based applications," in *ESEC-FSE companion '07: The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering.* New York, NY, USA: ACM, 2007, pp. 599–601.

[19] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 54–62, 1999.

[20] S. Vinoski, "A time for reflection," *IEEE Internet Computing*, vol. 9, no. 1, pp. 86–89, 2005.

[21] J. Dowling, T. Schafer, V. Cahill, P. Haraszti, and B. Redmond, "Using reflection to support dynamic adaptation of system software: A case study driven evaluation," in *OORaSE*, 1999, pp. 169–188. [Online]. Available: citeseer.ist.psu.edu/399915.html

[22] E. Bruneton, T. Coupaye, and J.-B. Stefani, "Fractal url," http://fractal.objectweb.org/documentation.html.

[23] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems," *Softw. Pract. Exper.*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[24] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng, "Composing adaptive software," *Computer*, vol. 37, no. 7, pp. 56–64, 2004.

[25] R. Calinescu, "Model-driven autonomic architecture," in *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing.* Washington, DC, USA: IEEE Computer Society, 2007, p. 9.

[26] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *FOSE '07: 2007 Future of Software Engineering.* Washington, DC, USA: IEEE Computer Society, 2007, pp. 259–268.

[27] K. Henricksen and J. Indulska, "Developing context-aware pervasive computing applications: Models and approach," *Pervasive and Mobile Computing*, vol. 2, no. 1, pp. 37–64, 2006.

[28] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "Towards architecture-based self-healing systems," in *WOSS '02: Proceedings of the first workshop on Self-healing systems.* New York, NY, USA: ACM, 2002, pp. 21–26.

[29] A. R. Butler, M. Ibrahim, K. Rennolls, and L. Bacon, "Composing simulation architectures for autonomic systems," *Knowl. Eng. Rev.*, vol. 21, no. 3, pp. 249–259, 2006.

[30] D. Garlan and B. Schmerl, “Model-based adaptation for self-healing systems,” in *WOSS ’02: Proceedings of the first workshop on Self-healing systems.* New York, NY, USA: ACM, 2002, pp. 27–32.

[31] S. Hassan, D. Mcsherry, and D. Bustard, “Autonomic self healing and recovery informed by environment knowledge,” *Artif. Intell. Rev.*, vol. 26, no. 1-2, pp. 89–101, 2006.

[32] C. Huebscher and A. McCann, “An adaptive middleware framework for context-aware applications,” *Personal Ubiquitous Comput.*, vol. 10, no. 1, pp. 12–20, 2005.

[33] W. Gilani, N. H. Naqvi, and O. Spinczyk, “On adaptable middleware product lines,” in *ARM ’04: Proceedings of the 3rd workshop on Adaptive and reflective middleware.* New York, NY, USA: ACM, 2004, pp. 207–213.

[34] P. Costa, G. Coulson, C. Mascolo, L. Mottola, G. P. Picco, and S. Zachariadis, “A reconfigurable component-based middleware for networked embedded systems,” *Journal of Wireless Information Networks*, vol. 14, no. 2, pp. 149–162, June 2007.

[35] T. Gu, H. K. Pung, and D. Q. Zhang, “A service-oriented middleware for building context-aware services,” *J. Netw. Comput. Appl.*, vol. 28, no. 1, pp. 1–18, 2005.

[36] R. Sterritt, M. Parashar, H. Tianfield, and R. Unland, “A concise introduction to autonomic computing,” *Advanced Engineering Informatics*, vol. 19, no. 3, pp. 181–187, July 2005. [Online]. Available: http://dx.doi.org/10.1016/j.aei.2005.05.012

[37] M. C. Huebscher and J. A. McCann, “A survey of autonomic computingdegrees, models, and applications,” *ACM Comput. Surv.*, vol. 40,

pp. 7:1–7:28, August 2008. [Online]. Available: http://doi.acm.org/10.1145/1380584.1380585

[38] T. O. Eze, R. J. Anthony, C. Walshaw, and A. Soper, "Autonomic computing in the first decade: Trends and direction," in *The Eighth International Conference on Autonomic and Autonomous Systems: ICAS 2012*. St. Maarten, The Netherlands Antilles: IARIA, March 2012, pp. 80–85.

[39] J. O. Kephart, "Autonomic computing: the first decade," in *Proceedings of the 8th ACM international conference on Autonomic computing*, ser. ICAC '11. New York, NY, USA: ACM, 2011, pp. 1–2. [Online]. Available: http://doi.acm.org/10.1145/1998582.1998584

[40] N. Damianou, N. Dulay, E. Lupu, and N. D. N. Dulay, "The ponder policy specification language," in *Lecture Notes in Computer Science*. Springer-Verlag, 2001, pp. 18–38.

[41] L. Lymberopoulos, E. Lupu, and M. Sloman, "An adaptive policy-based framework for network services management," *J. Netw. Syst. Manage.*, vol. 11, no. 3, pp. 277–303, 2003.

[42] R. J. Anthony, "The AGILE Policy Expression Language for Autonomic Systems," *ITSSA*, vol. 1, no. 4, pp. 381–398, 2006.

[43] DySCAS, "Dyscas project website," http://www.DySCAS.org.

[44] R. Anthony and C. Ekelin, "Policy-driven self-management for an automotive middleware," in *First International Workshop on Policy-Based Autonomic Computing (PBAC)*, Jacksonville, Florida, USA, June 2007.

[45] R. Anthony, A. Rettberg, I. Jahnich, M. Törngren, D. Chen, and C. Ekelin, "Towards a dynamically reconfigurable automotive control system architecture," in *International Embedded Systems Symposium*. Irvine, CA, USA: IFIP, May 2007.

[46] Richard Anthony, “Policy Autonomics website,” http://www.policyautonomics.net/.

[47] L. Kagal, “Rei : A Policy Language for the Me-Centric Project,” HP Labs, Tech. Rep., September 2002.

[48] Apache, “Imperius project,” http://incubator.apache.org/imperius/index.html.

[49] J. O. Kephart and R. Das, “Achieving self-management via utility functions,” *IEEE Internet Computing*, vol. 11, no. 1, pp. 40–48, 2007.

[50] R. Das, J. O. Kephart, J. Lenchner, and H. Hamann, “Utility-function-driven energy-efficient cooling in data centers,” in *Proceeding of the 7th international conference on Autonomic computing*, ser. ICAC ’10. New York, NY, USA: ACM, 2010, pp. 61–70. [Online]. Available: http://doi.acm.org/10.1145/1809049.1809058

[51] G. Tesauro, R. Das, W. E. Walsh, and J. O. Kephart, “Utility-function-driven resource allocation in autonomic systems,” *Autonomic Computing, International Conference on*, vol. 0, pp. 342–343, 2005.

[52] W. Walsh, G. Tesauro, J. Kephart, and R. Das, “Utility functions in autonomic systems,” in *Autonomic Computing, 2004. Proceedings. International Conference on*, may 2004, pp. 70 – 77.

[53] H. Li and S. Venugopal, “Using reinforcement learning for controlling an elastic web application hosting platform,” in *Proceedings of the 8th ACM international conference on Autonomic computing*, ser. ICAC ’11. New York, NY, USA: ACM, 2011, pp. 205–208. [Online]. Available: http://doi.acm.org/10.1145/1998582.1998630

[54] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1994.

[55] M. Al-Zawi, A. Hussain, D. Al-Jumeily, and A. Taleb-Bendiab, "Using adaptive neural networks in self-healing systems," in *Developments in eSystems Engineering (DESE), 2009 Second International Conference on*, dec. 2009, pp. 227 –232.

[56] B. Gaudin, E. I. Vassev, P. Nixon, and M. Hinchey, "A control theory based approach for self-healing of un-handled runtime exceptions," in *Proceedings of the 8th ACM international conference on Autonomic computing*, ser. ICAC '11. New York, NY, USA: ACM, 2011, pp. 217–220. [Online]. Available: http://doi.acm.org/10.1145/1998582.1998633

[57] J. Wildstrom, P. Stone, and E. Witchel, "Carve: A cognitive agent for resource value estimation." in *ICAC*, J. Strassner, S. A. Dobson, J. A. B. Fortes, and K. K. Goswami, Eds. IEEE Computer Society, 2008, pp. 182–191. [Online]. Available: http://dblp.uni-trier.de/db/conf/icac/icac2008.html#WildstromSW08

[58] A. Andrzejak, S. Graupner, and S. Plantikow, "Predicting resource demand in dynamic utility computing environments," in *Proceedings of the International Conference on Autonomic and Autonomous Systems*, ser. ICAS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 6–. [Online]. Available: http://dx.doi.org/10.1109/ICAS.2006.44

[59] A. Khalid, M. Haye, M. Khan, and S. Shamail, "Survey of frameworks, architectures and techniques in autonomic computing," in *Autonomic and Autonomous Systems, 2009. ICAS '09. Fifth International Conference on*, april 2009, pp. 220 –225.

[60] L. Capra and M. Musolesi, “Autonomic trust prediction for pervasive systems,” in *Advanced Information Networking and Applications, 2006. AINA 2006. 20th International Conference on*, vol. 2, April 2006.

[61] E. Kalyvianaki, T. Charalambous, and S. Hand, “Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters,” in *Proceedings of the 6th international conference on Autonomic computing*, ser. ICAC ’09. New York, NY, USA: ACM, 2009, pp. 117–126. [Online]. Available: http://doi.acm.org/10.1145/1555228.1555261

[62] T. Zheng, J. Yang, M. Woodside, M. Litoiu, and G. Iszlai, “Tracking time-varying parameters in software systems with extended kalman filters,” in *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, ser. CASCON ’05. IBM Press, 2005, pp. 334–345. [Online]. Available: http://dl.acm.org/citation.cfm?id=1105634.1105659

[63] M. Pelc, R. Anthony, and W. Byrski, “Policy-supervised exact state reconstruction in real-time embedded control systems,” in *Proceedings of ACD*, Zielona Gora, Poland, November 2009.

[64] J. O. Kephart, “Research challenges of autonomic computing,” in *ICSE ’05: Proceedings of the 27th international conference on Software engineering.* New York, NY, USA: ACM, 2005, pp. 15–22.

[65] M. Huebscher and J. McCann, “Evaluation issues in autonomic computing,” in *3rd international conference on grid and cooperative computing (GCC 2004), Wuhan, Peoples Republic of China*, 2004, p. 597. [Online]. Available: http://pubs.doc.ic.ac.uk/autonomic-eval/

[66] A. B. Brown and C. Redlin, “Measuring the effectiveness of self-healing autonomic systems,” *Autonomic Computing, International Conference on*, vol. 0, pp. 328–329, 2005.

[67] IBM, "An architectural blueprint for autonomic computing," IBM Whitepaper, 2004, http://www.ginkgo-networks.com/IMG/pdf/AC_Blueprint_White_Paper_V7.pdf.

[68] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky, "Validation, verification, and testing of computer software," *ACM Comput. Surv.*, vol. 14, no. 2, pp. 159–192, Jun. 1982. [Online]. Available: http://doi.acm.org/10.1145/356876.356879

[69] M. Kamel and S. Leue, "Vip: A visual editor and compiler for v-promela," in *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems.* London, UK: Springer-Verlag, 2000, pp. 471–486.

[70] S. Kikuchi, S. Tsuchiya, M. Adachi, and T. Katsuyama, "Policy verification and validation framework based on model checking approach," in *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing.* Washington, DC, USA: IEEE Computer Society, 2007, p. 1.

[71] J. Desel, A. Oberweis, T. Zimmer, and G. Zimmermann, "Validation of information system models: Petri nets and test case generation," in *Proc. of The 1997 IEEE International Conference On Systems, Man, And Cybernetics*, 1997, pp. 3401–3406.

[72] H. Huang and H. Kirchner, "Formal specification and verification of modular security policy based on colored petri nets," *IEEE Trans. on Dependable and Secure Computing*, vol. PP(99), pp. 1–13, 2010.

[73] J. Zhang and B. H. C. Cheng, "Model-based development of dynamically adaptive software," in *ICSE '06: Proceeding of the 28th international conference on Software engineering.* New York, NY, USA: ACM, 2006, pp. 371–380.

[74] T. D. Wolf and T. Holvoet, “A taxonomy for self-* propertes in decentralized autonomic computing,” *Autonomic Computing: Concepts, Infrastructure, and Applications*, vol. CRC Press, pp. 101–120, 2007.

[75] S. Lightstone, “Foundations of autonomic computing development,” in *Engineering of Autonomic and Autonomous Systems, 2007. EASe ’07. Fourth IEEE International Workshop on*, March 2007, pp. 163–171.

[76] V. Cortellessa, B. Cukic, D. D. Gobbo, A. Mili, M. Napolitano, M. Shereshevsky, and H. Sandhu, “Certifying adaptive flight control software,” in *IN PROCEEDINGS, ISACC 2000: THE SOFTWARE RISK MANAGEMENT CONFERENCE*, 2000.

[77] S. P. Miller, E. A. Anderson, L. G. Wagner, M. W. Whalen, and M. P. E. Heimdahl, “Formal verification of flight critical software.”

[78] E. S.-K. Yu, “Modelling strategic relationships for process reengineering,” Ph.D. dissertation, University of Toronto, Toronto, Ont., Canada, Canada, 1996, uMI Order No. GAXNN-02887 (Canadian dissertation).

[79] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos, “Tropos: An agent-oriented software development methodology,” *Autonomous Agents and Multi-Agent Systems*, vol. 8, no. 3, pp. 203–236, 2004.

[80] J. Castro, M. Kolp, and J. Mylopoulos, “Towards requirements-driven information systems engineering: the tropos project,” *Inf. Syst.*, vol. 27, no. 6, pp. 365–389, 2002.

[81] Y. Asnar, P. Giorgini, and J. Mylopoulos, “Goal-driven risk assessment in requirements engineering,” *Requirements Engineering*, vol. 16, pp. 101–116, 2011, 10.1007/s00766-010-0112-x. [Online]. Available: http://dx.doi.org/10.1007/s00766-010-0112-x

[82] P. Giorgini, J. Mylopoulos, and R. Sebastiani, “Goal-oriented requirements analysis and reasoning in the tropos methodology,” *Engineering Applications of Artificial Intelligence*, vol. 18, no. 2, pp. 159–171, March 2005.

[83] A. van Lamsweerde, “Goal-oriented requirements engineering: A roundtrip from research to practice,” *Requirements Engineering, IEEE International Conference on*, vol. 0, pp. 4–7, 2004.

[84] D. Amyot, S. Ghanavati, J. Horkoff, G. Mussbacher, L. Peyton, and E. Yu, “Evaluating goal models within the goal-oriented requirement language,” *Int. J. Intell. Syst.*, vol. 25, pp. 841–877, August 2010.

[85] A. Van Lamsweerde, “Goal-oriented requirements engineering: A guided tour,” in *RE ’01: Proceedings of the Fifth IEEE International Symposium on Requirements Engineering.* Washington, DC, USA: IEEE Computer Society, 2001, pp. 249–262.

[86] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2004.

[87] A. Sutcliffe, S. Thew, and P. Jarvis, “Experience with user-centred requirements engineering,” *Requirements Engineering*, vol. 16, pp. 267–280, 2011, 10.1007/s00766-011-0118-z. [Online]. Available: http://dx.doi.org/10.1007/s00766-011-0118-z

[88] I. J. Hayes, “Towards reasoning about teleo-reactive programs for robust real-time systems,” in *SERENE ’08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems.* New York, NY, USA: ACM, 2008, pp. 87–94.

[89] E. Gordon and B. Logan, “Game over: You have been beaten by a GRUE,” in *Challenges in Game Artificial Intelligence: Papers from the 2004 AAAI*

Workshop, D. Fu, S. Henke, and J. Orkin, Eds. AAAI Press, July 2004, pp. 16–21, Technical Report WS–04–04.

[90] E. P. Katz, "Extending the teleo-reactive paradigm for robotic agent task control using zadehan (fuzzy) logic," in *CIRA '97: Proceedings of the 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation.* Washington, DC, USA: IEEE Computer Society, 1997, p. 282.

[91] S. Coffey and K. Clark, "A Hybrid, Teleo-Reactive Architecture for Robot Control," in *Multi-Agent Robotic Systems*, August 2006. [Online]. Available: http://pubs.doc.ic.ac.uk/tr-robot-arch/

[92] C. Mcgann, F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. Mcewen, "T-rex: A model-based architecture for auv control," in *ICAP'07: 3rd Workshop on Planning and Plan Execution for Real-World Systems*, Rhode Island, USA, September 2007. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=?doi=10.1.1.120.3923

[93] G. Gubisch, G. Steinbauer, M. Weiglhofer, and F. Wotawa, "A teleo-reactive architecture for fast, reactive and robust control of mobile robots," in *New Frontiers in Applied Artificial Intelligence*, ser. Lecture Notes in Computer Science, N. Nguyen, L. Borzemski, A. Grzech, and M. Ali, Eds. Springer Berlin / Heidelberg, 2008, vol. 5027, pp. 541–550.

[94] B. Vargas and E. Morales, "Learning navigation teleo-reactive programs using behavioural cloning," in *Mechatronics, 2009. ICM 2009. IEEE International Conference on*, 14-17 2009, pp. 1 –6.

[95] M. J. Kochenderfer, "Evolving hierarchical and recursive teleo-reactive programs through genetic programming," in *Programming, EuroGP 2003, LNCS 2610.* Springer-Verlag, 2003, pp. 83–92.

[96] J. Ram0írez, "Neural synthesis of teleo-reactive programs," in *ICMAS '98: Proceedings of the 3rd International Conference on Multi Agent Systems.* Washington, DC, USA: IEEE Computer Society, 1998, p. 459.

[97] D. Choi and P. Langley, "Learning teleoreactive logic programs from problem solving," in *Proceedings of the Fifteenth International Conference on Inductive Logic Programming.* Springer, 2005, pp. 51–68.

[98] G. Riley, "Clips: An expert system building tool," in *Proceedings of the Technology 2001 Conference*, San Jose, CA, December 1991.

[99] E. Friedman, *Jess in action: rule-based systems in java.* Greenwich, CT, USA: Manning Publications Co., 2003.

[100] W. F. Clocksin and C. S. Mellish, *Programming in PROLOG; 1st ed.* Berlin: Springer, 1981.

[101] J. Hawthorne and R. Anthony, "Using a teleo-reactive approach in building self-managing systems," *International Journal of Autonomous and Adaptive Communication Systems*, vol. 5, no. 3, pp. 255–273, Jul. 2012. [Online]. Available: http://dx.doi.org/10.1504/IJAACS.2012.047658

[102] J. Hawthorne and R. Anthony, "A methodology for the use of the teleo-reactive programming technique in autonomic computing," in *Software Engineering Artificial Intelligence Networking and Parallel/Distributed Computing (SNPD), 2010 11th ACIS International Conference on*, June 2010, pp. 245 –250.

[103] J. Hawthorne and R. Anthony, "Developing teleo-reactive autonomic solutions," *International Journal of Computer & Information Science*, vol. 11, no. 3, pp. 26–34, 2010, international Journal of Computer & Information Science (IJCIS).

[104] K. Beck, *Test Driven Development: By Example.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[105] D. Astels, *Test Driven development: A Practical Guide.* Prentice Hall Professional Technical Reference, 2003.

[106] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[107] J. Hawthorne, R. Anthony, and M. Petridis, "Improving the development process for teleo-reactive programming through advanced composition," in *The Seventh International Conference on Autonomic and Autonomous Systems: ICAS 2011*, IARIA, Ed., Venice/Mestre, Italy, May 2011, pp. 75–80.

[108] C. Wang, K. Schwan, V. Talwar, G. Eisenhauer, L. Hu, and M. Wolf, "A flexible architecture integrating monitoring and analytics for managing large-scale data centers," in *Proceedings of the 8th ACM international conference on Autonomic computing*, ser. ICAC '11. New York, NY, USA: ACM, 2011, pp. 141–150. [Online]. Available: http://doi.acm.org/10.1145/1998582.1998605

[109] N. Vasic, T. Scherer, and W. Schott, "Thermal-aware workload scheduling for energy efficient data centers," in *Proceedings of the 7th international conference on Autonomic computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 169–174. [Online]. Available: http://doi.acm.org/10.1145/1809049.1809076

[110] T. Tashi, M. Hasan, and H. Yu, "A complete planner design of microstrip patch antenna for a passive uhf rfid tag," in *Automation and Computing (ICAC), 2011 17th International Conference on*, sept. 2011, pp. 12 –17.

[111] R. He, M. Lacoste, and J. Leneutre, "A policy management framework for self-protection of pervasive systems," in *Autonomic and Autonomous Systems (ICAS), 2010 Sixth International Conference on*, march 2010, pp. 104 –109.

[112] R. N. Charette, "Automated to death," http://spectrum.ieee.org/computing/software/automated-to-death/0, December 2009.

[113] L. Bainbridge, "Ironies of automation," *Automatica*, vol. 19, no. 6, pp. 775 – 779, 1983. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0005109883900468

[114] D. Andre, "Artificial evolution of intelligence: Lessons from natural evolution: An illustrative approach using genetic programming," 1994.

[115] P. Srinivasan, "Development of block-stacking teleo-reactive programs using genetic programming," in *Genetic Algorithms and Genetic Programming at Stanford 2002*, J. R. Koza, Ed. Stanford, California, 94305-3079 USA: Stanford Bookstore, June 2002, pp. 233–242. [Online]. Available: http://www.genetic-programming.org/sp2002/Srinivasan.pdf

[116] J. Ram0írez, "Teleoreactive neural networks," in *Biological and Artificial Computation: From Neuroscience to Technology*, ser. Lecture Notes in Computer Science, J. Mira, R. Moreno-Daz, and J. Cabestany, Eds. Springer Berlin / Heidelberg, 1997, vol. 1240, pp. 1384–1393, 10.1007/BFb0032599. [Online]. Available: http://dx.doi.org/10.1007/BFb0032599

[117] S. S. Benson, "Learning action models for reactive autonomous agents," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1997.

[118] M. R. K. Ryan and M. D. Pendrith, "Rl-tops: An architecture for modularity and re-use in reinforcement learning," in *Proceedings of the Fifteenth*

International Conference on Machine Learning, ser. ICML '98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 481–487. [Online]. Available: http://dl.acm.org/citation.cfm?id=645527.657454

[119] Linux, "Ubuntu," http://www.ubuntu.com/.

[120] Mozilla, "Firefox," http://www.mozilla.org/en-US/firefox/new/.

[121] Apache, "Openoffice," http://www.openoffice.org/.

APPENDIX A

DATACENTRE SCRIPTS

A.1 requests.xml

```
<?xml version="1.0" encoding="utf-8"?>
<EventScript>
      <AddAppEvent>
            <AppName>app1</AppName>
            <Offset>0</Offset>
      </AddAppEvent>
      <AddAppEvent>
            <AppName>app2</AppName>
            <Offset>0</Offset>
      </AddAppEvent>
      <AddAppEvent>
            <AppName>service1</AppName>
            <Offset>0</Offset>
      </AddAppEvent>
      <AddAppEvent>
            <AppName>service2</AppName>
            <Offset>0</Offset>
      </AddAppEvent>

      <BandedEvents>
            <AppName>app1</AppName>
            <StartOffset>1000</StartOffset>
            <EndOffset>360000</EndOffset>
            <PacketSize>15000</PacketSize>
            <Frequency>60000</Frequency>
            <InitialSize>15000</InitialSize>
            <FinalSize>275000</FinalSize>
            <PlusMinusBand>20</PlusMinusBand>
      </BandedEvents>
            <BandedEvents>
            <AppName>app2</AppName>
            <StartOffset>1000</StartOffset>
            <EndOffset>360000</EndOffset>
            <PacketSize>15000</PacketSize>
            <Frequency>60000</Frequency>
            <InitialSize>15000</InitialSize>
            <FinalSize>100000</FinalSize>
            <PlusMinusBand>20</PlusMinusBand>
      </BandedEvents>
            <BandedEvents>
            <AppName>service1</AppName>
            <StartOffset>1000</StartOffset>
            <EndOffset>360000</EndOffset>
            <PacketSize>15000</PacketSize>
            <Frequency>60000</Frequency>
            <InitialSize>15000</InitialSize>
            <FinalSize>400000</FinalSize>
            <PlusMinusBand>20</PlusMinusBand>
      </BandedEvents>
            <BandedEvents>
            <AppName>service2</AppName>
            <StartOffset>1000</StartOffset>
            <EndOffset>360000</EndOffset>
            <PacketSize>15000</PacketSize>
```

```
        <Frequency>60000</Frequency>
        <InitialSize>15000</InitialSize>
        <FinalSize>800000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>

<BandedEvents>
        <AppName>app1</AppName>
        <StartOffset>360000</StartOffset>
        <EndOffset>720000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>275000</InitialSize>
        <FinalSize>400000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>app2</AppName>
        <StartOffset>360000</StartOffset>
        <EndOffset>720000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>100000</InitialSize>
        <FinalSize>300000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>service1</AppName>
        <StartOffset>360000</StartOffset>
        <EndOffset>720000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>400000</InitialSize>
        <FinalSize>420000</FinalSize>
```

```
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>service2</AppName>
      <StartOffset>360000</StartOffset>
      <EndOffset>720000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>800000</InitialSize>
      <FinalSize>600000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>

<BandedEvents>
      <AppName>app1</AppName>
      <StartOffset>720000</StartOffset>
      <EndOffset>1080000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>400000</InitialSize>
      <FinalSize>500000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>app2</AppName>
      <StartOffset>720000</StartOffset>
      <EndOffset>1080000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>300000</InitialSize>
      <FinalSize>400000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>service1</AppName>
      <StartOffset>720000</StartOffset>
      <EndOffset>1080000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>420000</InitialSize>
      <FinalSize>420000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>service2</AppName>
      <StartOffset>720000</StartOffset>
      <EndOffset>1080000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>600000</InitialSize>
      <FinalSize>700000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>

<BandedEvents>
      <AppName>app1</AppName>
      <StartOffset>1080000</StartOffset>
      <EndOffset>1440000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>500000</InitialSize>
      <FinalSize>150000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>app2</AppName>
      <StartOffset>1080000</StartOffset>
      <EndOffset>1440000</EndOffset>
```

```
            <PacketSize >15000</PacketSize >
            <Frequency >60000</Frequency >
            <InitialSize >400000</InitialSize >
            <FinalSize >50000</FinalSize >
            <PlusMinusBand >20</PlusMinusBand >
      </BandedEvents >
            <BandedEvents >
            <AppName >service1 </AppName >
            <StartOffset >1080000</StartOffset >
            <EndOffset >1440000</EndOffset >
            <PacketSize >15000</PacketSize >
            <Frequency >60000</Frequency >
            <InitialSize >420000</InitialSize >
            <FinalSize >200000</FinalSize >
            <PlusMinusBand >20</PlusMinusBand >
      </BandedEvents >
            <BandedEvents >
            <AppName >service2 </AppName >
            <StartOffset >1080000</StartOffset >
            <EndOffset >1440000</EndOffset >
            <PacketSize >15000</PacketSize >
            <Frequency >60000</Frequency >
            <InitialSize >700000</InitialSize >
            <FinalSize >250000</FinalSize >
            <PlusMinusBand >20</PlusMinusBand >
      </BandedEvents >

      <BandedEvents >
            <AppName >app1 </AppName >
            <StartOffset >1440000</StartOffset >
            <EndOffset >1800000</EndOffset >
            <PacketSize >15000</PacketSize >
            <Frequency >60000</Frequency >
            <InitialSize >150000</InitialSize >
```

```
        <FinalSize>100000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>app2</AppName>
        <StartOffset>1440000</StartOffset>
        <EndOffset>1800000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>50000</InitialSize>
        <FinalSize>15000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>service1</AppName>
        <StartOffset>1440000</StartOffset>
        <EndOffset>1800000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>200000</InitialSize>
        <FinalSize>500000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>service2</AppName>
        <StartOffset>1440000</StartOffset>
        <EndOffset>1800000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>250000</InitialSize>
        <FinalSize>20000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
```

```
<BandedEvents>
      <AppName>app1</AppName>
      <StartOffset>1800000</StartOffset>
      <EndOffset>2160000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>100000</InitialSize>
      <FinalSize>400000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>app2</AppName>
      <StartOffset>1800000</StartOffset>
      <EndOffset>2160000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>15000</InitialSize>
      <FinalSize>300000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>service1</AppName>
      <StartOffset>1800000</StartOffset>
      <EndOffset>2160000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>500000</InitialSize>
      <FinalSize>600000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>service2</AppName>
      <StartOffset>1800000</StartOffset>
      <EndOffset>2160000</EndOffset>
```

```
            <PacketSize>15000</PacketSize>
            <Frequency>60000</Frequency>
            <InitialSize>20000</InitialSize>
            <FinalSize>100000</FinalSize>
            <PlusMinusBand>20</PlusMinusBand>
      </BandedEvents>
</EventScript>
```

A.2 trendingdata.xml

```
<?xml version="1.0" encoding="utf-8"?>
<EventScript>
      <AddAppEvent>
            <AppName>app1</AppName>
            <Offset>0</Offset>
      </AddAppEvent>
      <AddAppEvent>
            <AppName>app2</AppName>
            <Offset>0</Offset>
      </AddAppEvent>
      <AddAppEvent>
            <AppName>service1</AppName>
            <Offset>0</Offset>
      </AddAppEvent>
      <AddAppEvent>
            <AppName>service2</AppName>
            <Offset>0</Offset>
      </AddAppEvent>

      <BandedEvents>
            <AppName>app1</AppName>
            <StartOffset>1000</StartOffset>
            <EndOffset>300000</EndOffset>
            <PacketSize>15000</PacketSize>
            <Frequency>60000</Frequency>
            <InitialSize>15000</InitialSize>
            <FinalSize>25000</FinalSize>
            <PlusMinusBand>20</PlusMinusBand>
      </BandedEvents>
            <BandedEvents>
            <AppName>app2</AppName>
            <StartOffset>1000</StartOffset>
            <EndOffset>300000</EndOffset>
            <PacketSize>15000</PacketSize>
            <Frequency>60000</Frequency>
            <InitialSize>15000</InitialSize>
            <FinalSize>100000</FinalSize>
            <PlusMinusBand>20</PlusMinusBand>
      </BandedEvents>
            <BandedEvents>
            <AppName>service1</AppName>
            <StartOffset>1000</StartOffset>
            <EndOffset>300000</EndOffset>
            <PacketSize>15000</PacketSize>
            <Frequency>60000</Frequency>
            <InitialSize>15000</InitialSize>
            <FinalSize>40000</FinalSize>
            <PlusMinusBand>20</PlusMinusBand>
      </BandedEvents>
            <BandedEvents>
            <AppName>service2</AppName>
            <StartOffset>1000</StartOffset>
            <EndOffset>300000</EndOffset>
            <PacketSize>15000</PacketSize>
            <Frequency>60000</Frequency>
            <InitialSize>15000</InitialSize>
            <FinalSize>25000</FinalSize>
            <PlusMinusBand>20</PlusMinusBand>
      </BandedEvents>
```

```
<BandedEvents>
      <AppName>app1</AppName>
      <StartOffset>300000</StartOffset>
      <EndOffset>430000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>25000</InitialSize>
      <FinalSize>125000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>app2</AppName>
      <StartOffset>300000</StartOffset>
      <EndOffset>430000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>100000</InitialSize>
      <FinalSize>200000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>service1</AppName>
      <StartOffset>300000</StartOffset>
      <EndOffset>430000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>40000</InitialSize>
      <FinalSize>42000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>service2</AppName>
      <StartOffset>300000</StartOffset>
```

```
        <EndOffset>430000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>25000</InitialSize>
        <FinalSize>125000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>

<BandedEvents>
        <AppName>app1</AppName>
        <StartOffset>430000</StartOffset>
        <EndOffset>600000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>125000</InitialSize>
        <FinalSize>137500</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>app2</AppName>
        <StartOffset>430000</StartOffset>
        <EndOffset>600000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>200000</InitialSize>
        <FinalSize>250000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>service1</AppName>
        <StartOffset>430000</StartOffset>
        <EndOffset>600000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>42000</InitialSize>
        <FinalSize>60000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>service2</AppName>
        <StartOffset>430000</StartOffset>
        <EndOffset>600000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>125000</InitialSize>
        <FinalSize>137000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>

<BandedEvents>
        <AppName>app1</AppName>
        <StartOffset>600000</StartOffset>
        <EndOffset>800000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>137500</InitialSize>
        <FinalSize>350000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>app2</AppName>
        <StartOffset>600000</StartOffset>
        <EndOffset>800000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>250000</InitialSize>
        <FinalSize>450000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
```

```
</BandedEvents>
      <BandedEvents>
      <AppName>service1</AppName>
      <StartOffset>600000</StartOffset>
      <EndOffset>800000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>60000</InitialSize>
      <FinalSize>120000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>service2</AppName>
      <StartOffset>600000</StartOffset>
      <EndOffset>800000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>137000</InitialSize>
      <FinalSize>350000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>

<BandedEvents>
      <AppName>app1</AppName>
      <StartOffset>800000</StartOffset>
      <EndOffset>1300000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>350000</InitialSize>
      <FinalSize>362500</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>app2</AppName>
      <StartOffset>800000</StartOffset>
      <EndOffset>1300000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>450000</InitialSize>
      <FinalSize>600000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>service1</AppName>
      <StartOffset>800000</StartOffset>
      <EndOffset>1300000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>120000</InitialSize>
      <FinalSize>250000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>service2</AppName>
      <StartOffset>800000</StartOffset>
      <EndOffset>1300000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>350000</InitialSize>
      <FinalSize>362500</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>

<BandedEvents>
      <AppName>app1</AppName>
      <StartOffset>1300000</StartOffset>
      <EndOffset>1600000</EndOffset>
      <PacketSize>15000</PacketSize>
```

```
        <Frequency>60000</Frequency>
        <InitialSize>362500</InitialSize>
        <FinalSize>600000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>app2</AppName>
        <StartOffset>1300000</StartOffset>
        <EndOffset>1600000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>600000</InitialSize>
        <FinalSize>800000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>service1</AppName>
        <StartOffset>1300000</StartOffset>
        <EndOffset>1600000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>250000</InitialSize>
        <FinalSize>400000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>service2</AppName>
        <StartOffset>1300000</StartOffset>
        <EndOffset>1600000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>362500</InitialSize>
        <FinalSize>600000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
```

```
</BandedEvents>


<BandedEvents>
      <AppName>app1</AppName>
      <StartOffset>1600000</StartOffset>
      <EndOffset>2000000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>600000</InitialSize>
      <FinalSize>500000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>app2</AppName>
      <StartOffset>1600000</StartOffset>
      <EndOffset>2000000</EndOffset>0
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>800000</InitialSize>
      <FinalSize>650000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>service1</AppName>
      <StartOffset>1600000</StartOffset>
      <EndOffset>2000000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>400000</InitialSize>
      <FinalSize>300000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>service2</AppName>
```

```
        <StartOffset>1600000</StartOffset>
        <EndOffset>2000000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>600000</InitialSize>
        <FinalSize>500000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>

<BandedEvents>
        <AppName>app1</AppName>
        <StartOffset>2000000</StartOffset>
        <EndOffset>2200000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>500000</InitialSize>
        <FinalSize>400000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>app2</AppName>
        <StartOffset>2000000</StartOffset>
        <EndOffset>2200000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>650000</InitialSize>
        <FinalSize>600000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>service1</AppName>
        <StartOffset>2000000</StartOffset>
        <EndOffset>2200000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>300000</InitialSize>
        <FinalSize>250000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>service2</AppName>
        <StartOffset>2000000</StartOffset>
        <EndOffset>2200000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>500000</InitialSize>
        <FinalSize>400000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>

<BandedEvents>
        <AppName>app1</AppName>
        <StartOffset>2200000</StartOffset>
        <EndOffset>2500000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>400000</InitialSize>
        <FinalSize>350000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>app2</AppName>
        <StartOffset>2200000</StartOffset>
        <EndOffset>2500000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>600000</InitialSize>
        <FinalSize>500000</FinalSize>
```

```
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>service1</AppName>
      <StartOffset>2200000</StartOffset>
      <EndOffset>2500000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>1
      <InitialSize>250000</InitialSize>
      <FinalSize>200000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>service2</AppName>
      <StartOffset>2200000</StartOffset>
      <EndOffset>2500000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>400000</InitialSize>
      <FinalSize>350000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>

<BandedEvents>
      <AppName>app1</AppName>
      <StartOffset>2500000</StartOffset>
      <EndOffset>2900000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>350000</InitialSize>
      <FinalSize>200000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>app2</AppName>
      <StartOffset>2500000</StartOffset>
      <EndOffset>2900000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>500000</InitialSize>
      <FinalSize>400000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>service1</AppName>
      <StartOffset>2500000</StartOffset>
      <EndOffset>2900000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>200000</InitialSize>
      <FinalSize>100000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
      <BandedEvents>
      <AppName>service2</AppName>
      <StartOffset>2500000</StartOffset>
      <EndOffset>2900000</EndOffset>
      <PacketSize>15000</PacketSize>
      <Frequency>60000</Frequency>
      <InitialSize>350000</InitialSize>
      <FinalSize>200000</FinalSize>
      <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>

<BandedEvents>
      <AppName>app1</AppName>
      <StartOffset>2900000</StartOffset>
      <EndOffset>3200000</EndOffset>
```

```
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>200000</InitialSize>
        <FinalSize>100000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>app2</AppName>
        <StartOffset>2900000</StartOffset>
        <EndOffset>3200000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>400000</InitialSize>
        <FinalSize>300000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>service1</AppName>
        <StartOffset>2900000</StartOffset>
        <EndOffset>3200000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>100000</InitialSize>
        <FinalSize>80000</FinalSize>
        <PlusMinusBand>20</PlusMinusBand>
</BandedEvents>
        <BandedEvents>
        <AppName>service2</AppName>
        <StartOffset>2900000</StartOffset>
        <EndOffset>3200000</EndOffset>
        <PacketSize>15000</PacketSize>
        <Frequency>60000</Frequency>
        <InitialSize>200000</InitialSize>
        <FinalSize>100000</FinalSize>
```

```
            <PlusMinusBand>20</PlusMinusBand>
      </BandedEvents>
</EventScript>
```

APPENDIX B

UNIT TESTS

This section contains a small selection of unit tests which were used in development of JTRAF and the use-case examples. They provide documentation for the work and usage examples.

B.1 JTRAF Tests

B.1.1 EmptyTRProg.java

```
package tests.mockClasses;

import trProgram.Conditional;
import trProgram.TRProgram;

public class EmptyTRProg extends TRProgram
{
    public EmptyTRProg(Conditional condition)
    {
      super(condition);
    }

    public EmptyTRProg(){}
}
```

B.1.2 TRProgTest.java

```
package tests;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
import tests.mockClasses.EmptyTRProg;
import trProgram.*;

public class TRProgTest
{

    private EmptyTRProg _trProg;

    @Before
    public void setUp()
    {
      _trProg = new EmptyTRProg();
    }

    @Test
    public void cannotInsert2SameConditions()
    {
      //same condition can only be added once

      Conditional first = new IsFalse();
      assertTrue(_trProg.addCondition(first));

      Conditional second = first;
      assertFalse(_trProg.addCondition(second));
    }
```

```
@Test
public void trueCondition()
{
  Conditional trueCondition = new IsTrue();
  _trProg.addCondition(trueCondition);

  assertTrue(_trProg.getConditionChecker()
        .testCondition(trueCondition));
  assertEquals("T", trueCondition.toString());
}

@Test
public void notCondition()
{
  Conditional notCondition = new NotCondition(new IsTrue());
  _trProg.addCondition(notCondition);

  assertFalse(_trProg.getConditionChecker()
        .testCondition(notCondition));
  assertEquals("!T", notCondition.toString());
  assertEquals(2, _trProg.conditionCount());
}

@Test
public void simpleAndFalse()
{
  Conditional andCondition =
        new AndCondition(new IsTrue(), new IsFalse());
  _trProg.addCondition(andCondition);

  assertFalse(_trProg.getConditionChecker()
        .testCondition(andCondition));
  assertEquals("T && F", andCondition.toString());
  assertEquals(3, _trProg.conditionCount());
```

```
}

@Test
public void simpleAndTrue()
{
  Conditional andCondition =
        new AndCondition(new IsTrue(), new NotCondition(
        new IsFalse()));
  _trProg.addCondition(andCondition);

  assertTrue(_trProg.getConditionChecker()
        .testCondition(andCondition));
  assertEquals("T && !F", andCondition.toString());
  assertEquals(4, _trProg.conditionCount());
}

@Test
public void complexCondition()
{
  // True && (False || !False)

  Conditional trueCondition = new IsTrue();
  Conditional falseCondition = new IsFalse();
  Conditional notCondition = new NotCondition(falseCondition);
  Conditional orCondition = new OrCondition(
        falseCondition, notCondition);
  Conditional andCondition = new AndCondition(
        trueCondition, orCondition);
  _trProg.addCondition(andCondition);

  assertTrue(_trProg.getConditionChecker()
        .testCondition(andCondition));
  assertEquals("T && F || !F", andCondition.toString());
```

```
    //There should only be 5 conditions since false used twice
    assertEquals(5, _trProg.conditionCount());
  }

  @Test
  public void getConcreteConditions()
  {
    Conditional concreteCondition = new IsTrue();
    _trProg.addCondition(concreteCondition);

    assertEquals(1, _trProg.getConcreteConditions().size());
    assertTrue(_trProg.getConcreteConditions()
          .contains(concreteCondition));
  }

  @Test
  public void getConcreteNoted()
  {
    Conditional concreteCondition = new IsTrue();
    _trProg.addCondition(new NotCondition(concreteCondition));

    assertEquals(1, _trProg.getConcreteConditions().size());
    assertTrue(_trProg.getConcreteConditions()
          .contains(concreteCondition));
  }

  @Test
  public void getConreteComplex()
  {
    Conditional trueCondition = new IsTrue();
    Conditional falseCondition = new IsFalse();
    Conditional notCondition = new NotCondition(falseCondition);
    Conditional orCondition = new OrCondition(
          falseCondition, notCondition);
```

```
      Conditional andCondition = new AndCondition(
            trueCondition, orCondition);
      _trProg.addCondition(andCondition);

      assertEquals(2, _trProg.getConcreteConditions().size());
      assertTrue(_trProg.getConcreteConditions()
            .contains(trueCondition));
      assertTrue(_trProg.getConcreteConditions()
            .contains(falseCondition));
    }
}
```

B.1.3 MockAction.java

```
package tests.mockClasses;

import trProgram.Action;
import trProgram.Conditional;
import trProgram.TRProgram;

public class MockAction extends Action
{

    private String _output;
    private String _name;

    public MockAction(Conditional condition)
    {
      this(condition, "not set");
    }

    public MockAction(Conditional condition, String name)
    {
      super(condition);
      _output = "Do Nothing!";
      _name = name;
    }

    //main method to override
    public void action(Object parameters)
    {
      _output = "Do Something!";
    }

    public String getOutput()
    {
      return _output;
    }

    public void setName(String name)
    {
      _name = name;
    }

    @Override
    public String toString()
    {
      return _name;
    }
}
```

B.1.4 MockActionTest.java

```
package tests;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Ignore;
import org.junit.Test;
import tests.mockClasses.EmptyTRProg;
import tests.mockClasses.MockAction;
import trProgram.*;

public class MockActionTest
{

    private EmptyTRProg _emptyProg;
    private Conditional _trueCondition;
    private MockAction _returnTextAction;

    @Before
    public void setUp()
    {
      _emptyProg = new EmptyTRProg();
      _trueCondition = new IsTrue();
      _returnTextAction = new MockAction(_trueCondition);
    }

    @Test(expected = IllegalArgumentException.class)
    public void actionNotAdded()
    {
      _emptyProg.runAction(_returnTextAction);
    }

    @Ignore("No longer needed because condition
        is automatically added when adding the action")
    @Test(expected = IllegalArgumentException.class)
    public void actionAddedButConditionNot()
    {
      _emptyProg.addActionToBottom(_returnTextAction);
      _emptyProg.runAction(_returnTextAction);
    }
    @Test
    public void actionAndConditionAdded()
    {
      _emptyProg.addCondition(_trueCondition);
      _emptyProg.addActionToBottom(_returnTextAction);
      _emptyProg.runAction(_returnTextAction);

      assertEquals("Do Something!", _returnTextAction.getOutput());
    }

    @Test
    public void actionAndConditionAddedDoesSomething()
    {
      _emptyProg.addActionToBottom(_returnTextAction);
      _emptyProg.runAction(_returnTextAction);

      assertEquals("Do Something!", _returnTextAction.getOutput());
    }
}
```

B.2 File Sender Tests

B.2.1 ConnectActionTests.java

```
package tests;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

import trProgram.Action;
import trProgram.AndCondition;
import trProgram.Conditional;
import concreteNetExample.Client;
import concreteNetExample.TRProgNet;
import concreteNetExample.Actions.NetActionFactory;
import concreteNetExample.Conditions.AndConditionPassRHS;
import concreteNetExample.Conditions.NetConditionFactory;

public class ConnectActionTest
{

    private TRProgNet _trProg;
    private Conditional _isAccepting;
    private Conditional _isWaiting;
    private Conditional _isConnected;
    private Action _connectAction;

    @Before
    public void setup()
    {
      _trProg = new TRProgNet();
      _isAccepting = NetConditionFactory
            .createCondition(_trProg, "IsAccepting");
      _isWaiting = NetConditionFactory
            .createCondition(_trProg, "IsWaiting");
      _isConnected = NetConditionFactory
            .createCondition(_trProg, "IsConnected");
      _connectAction = NetActionFactory.createAction(
            new AndConditionPassRHS(_isAccepting, _isWaiting),
            "ConnectAction");
      _trProg.addCondition(_isConnected);
      _trProg.addActionToBottom(_connectAction);
    }

    @Test
```

```
public void connectActionNotAccepting()
{
  _trProg.getSocket().setAccept(false);
  _trProg.getSocket().connect(new Client(6000));
  _trProg.singleRun();

  assertFalse(_trProg.getConditionChecker()
        .testCondition(_isConnected));
}

@Test
public void connectActionAccepting()
{
  _trProg.getSocket().setAccept(true);
  _trProg.getSocket().connect(new Client(6000));
  _trProg.singleRun();

  assertTrue(_trProg.getConditionChecker()
        .testCondition(_isConnected));
}

@Test
public void connectActionAcceptingNotWaiting()
{
  _trProg.getSocket().setAccept(true);
  _trProg.runAction(_connectAction);

  assertFalse(_trProg.getConditionChecker()
        .testCondition(_isConnected));
}

@Test
public void connectAcceptWaitingFalse()
{
        _trProg.getSocket().setAccept(true);
        _trProg.getSocket().connect(new Client(6000));
        _trProg.runAction(_connectAction);

        assertTrue(_trProg.getConditionChecker()
              .testCondition(_isConnected));
        assertFalse("After connecting, should stop accepting",
              _trProg.getConditionChecker()
              .testCondition(_isAccepting));
        assertFalse("After connecting,
        there should be no-one waiting",
              _trProg.getConditionChecker()
              .testCondition(_isWaiting));
    }
}
```

B.2.2 FullClientTest.java

```
package tests;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

import trProgram.Action;
import trProgram.AndCondition;
import trProgram.Conditional;
import trProgram.IsTrue;
import trProgram.Nil;
import concreteNetExample.Client;
import concreteNetExample.TRProgNet;
import concreteNetExample.Actions.NetActionFactory;
import concreteNetExample.Conditions.AndConditionPassRHS;
import concreteNetExample.Conditions.IsTrueNet;
import concreteNetExample.Conditions.NetConditionFactory;

public class FullClientSendTest
{

    private TRProgNet _trProg;
    private Conditional _istrue, _isAccepting, _isWaiting,
          _isConnected, _is5PacketsSent, _isFileComplete;
    private Action _acceptAction, _connectAction, _sendPacket,
          _changePacketSize, _nil;

    @Before
    public void setup()
    {
      _trProg = new TRProgNet(5);
      _istrue = new IsTrueNet(_trProg);
      _isAccepting = NetConditionFactory
            .createCondition(_trProg, "IsAccepting");
      _isWaiting = NetConditionFactory
            .createCondition(_trProg, "IsWaiting");
      _isConnected = NetConditionFactory
            .createCondition(_trProg, "IsConnected");
      _is5PacketsSent = NetConditionFactory
            .createCondition(_trProg, "IsFivePacketsSent");
      _isFileComplete = NetConditionFactory
            .createCondition(_trProg, "IsFileComplete");

      _acceptAction = NetActionFactory
            .createAction(_istrue, "AcceptAction");
      _connectAction = NetActionFactory.createAction(
            new AndConditionPassRHS(
            _isAccepting, _isWaiting), "ConnectAction");
      _sendPacket = NetActionFactory.createAction(
            _isConnected, "SendNextPacketAction");
      _changePacketSize = NetActionFactory.createAction(
            _is5PacketsSent, _sendPacket, "ChangePacketSizeAction");
      _nil = new Nil(_isFileComplete);

      _trProg.addActionToBottom(_nil);
      _trProg.addActionToBottom(_changePacketSize);
      _trProg.addActionToBottom(_sendPacket);
      _trProg.addActionToBottom(_connectAction);
      _trProg.addActionToBottom(_acceptAction);
    }

    @Test
    public void fullTestNoThreads()
    {
      for (int i = 0; i < 2; i++)
      {
          _trProg.singleRun();
      }

      _trProg.getSocket().connect(new Client(5000));

      for (int i = 0; i < 65; i++)
      {
          _trProg.singleRun();
      }

      assertTrue(_trProg.getConditionChecker()
            .testCondition(_isFileComplete));
    }

    @Test
    public void runKillNotComplete()
    {
      Thread trProgThread = new Thread(_trProg);

      trProgThread.start();

      try
      {
          Thread.sleep(300);
      } catch (InterruptedException e)
      {
      }

      _trProg.kill();

      assertFalse(_trProg.getConditionChecker()
            .testCondition(_isFileComplete));
    }

    @Test
    public void runConnectClientComplete()
    {
      Thread trProgThread = new Thread(_trProg);

      trProgThread.start();

      try
      {
          Thread.sleep(100);
      } catch (InterruptedException e)
      {
      }

      _trProg.getSocket().connect(new Client(5000));

      try
```

```
      {
          Thread.sleep(500);
      } catch (InterruptedException e)
      {
      }

      _trProg.kill();

      assertTrue(_trProg.getConditionChecker()
            .testCondition(_isFileComplete));
    }
}
```

B.3 Datacentre Tests

B.3.1 DatacentreTests.java

```
package tests;

import datacentre.SimplePubSubTimer;
import datacentre.model.Application;
import datacentre.model.DatacentreModel;
import datacentre.model.Event;
import datacentre.model.RequestEvent;
import datacentre.model.EventHandler;
import datacentre.model.Pool;
import datacentre.model.Server;
import java.util.ArrayList;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.junit.Before;
import org.junit.Ignore;
import org.junit.Test;
import static org.junit.Assert.*;
```

```
public class DatacentreTest
{

    private DatacentreModel _datacentre;

    @Before
    public void setup()
    {
      _datacentre = new DatacentreModel();
    }

    @Test
    public void noApps()
    {
      assertEquals(0, _datacentre.appsServed());
    }

    @Test
    public void oneApp()
    {
      _datacentre.addApp(new Application("database 1"));

      assertEquals(1, _datacentre.appsServed());
    }

    @Test
    public void twoAppsSame()
    {
      _datacentre.addApp(new Application("database 1"));
      _datacentre.addApp(new Application("database 1"));

      assertEquals(1, _datacentre.appsServed());
    }
```

```
@Test
public void removeApp()
{
  Application app = new Application("database 1");

  _datacentre.addApp(app);
  _datacentre.removeApp(new Application("nonexistant app"));

  assertEquals(1, _datacentre.appsServed());

  _datacentre.removeApp(app);

  assertEquals(0, _datacentre.appsServed());
}

@Test
public void noPools()
{
  assertEquals(0, _datacentre.getPools().size());
}

@Test
public void noAppNoEvent()
{
  EventHandler eventHandler = new EventHandler();
  ArrayList<Event> events = new ArrayList<Event>();
  ArrayList<Event> oldEvents = new ArrayList<Event>();
  ArrayList<Event> executionEvents = eventHandler
      .getExecutionList(60, events, oldEvents);

  assertTrue(executionEvents.isEmpty());
}
```

```
@Test
public void provisionServer()
{
  Pool pool = Pool.createPool(new Application("null"));
  Server server = new Server(350);
  _datacentre.addIdleServer(server);
  _datacentre.addIdleServer(new Server(350));
  _datacentre.addIdleServer(new Server(350));
  _datacentre.addPool(pool);
  _datacentre.provision(server, pool);

  assertEquals(2, _datacentre.getIdleServers().size());
  assertEquals(1, _datacentre.getprovisioningServers().size());
  assertTrue(server.getProvisionStamp() > 0);
  assertEquals(pool, server.getProvisionPool());
}

@Test
public void removePoolreturnServers()
{
  Pool pool = Pool.createPool(new Application("null"));
  Server server = new Server(350);
  _datacentre.addIdleServer(server);
  _datacentre.addIdleServer(new Server(350));
  _datacentre.addIdleServer(new Server(350));
  _datacentre.addPool(pool);
  _datacentre.provision(server, pool);
  _datacentre.removePool(pool);

  assertEquals(3, _datacentre.getIdleServers().size());
  assertEquals(0, _datacentre.getprovisioningServers().size());
  assertEquals(0, server.getProvisionStamp());
}
```

```
    @Test
    public void serversForPool()
    {
      Pool pool = Pool.createPool(new Application("null"));
      Pool pool2 = Pool.createPool(new Application("app"));
      _datacentre.addPool(pool);
      _datacentre.addPool(pool2);
      Server server1 = new Server(350);
      Server server2 = new Server(350);
      Server server3 = new Server(350);
      Server server5 = new Server(350);
      Server server4 = new Server(350);
      _datacentre.addIdleServer(server1);
      _datacentre.addIdleServer(server2);
      _datacentre.addIdleServer(server3);
      _datacentre.addIdleServer(server4);
      _datacentre.addIdleServer(server5);
      _datacentre.provision(server1, pool);
      _datacentre.provision(server2, pool);
      _datacentre.provision(server3, pool);
      _datacentre.provision(server4, pool2);

      assertEquals(3, _datacentre
          .getServersProvisionedForPool(pool).size());
    }
}
```

B.3.2 OverloadedTest.java

```
package tests;

import datacentre.manager.Overloaded;
import datacentre.manager.Provisioner;
import datacentre.model.Application;
import datacentre.model.DatacentreModel;
import datacentre.model.Pool;
import datacentre.model.Request;
import datacentre.model.Server;
import org.junit.Before;
import org.junit.Test;
import trProgram.Conditional;
import trProgram.TRProgram;
import static org.junit.Assert.*;

public class OverloadedTest
{

    private DatacentreModel _model;
    private TRProgram _provisioner;
    private Application _app;
    private Conditional _overloaded;
    private Pool _pool;

    @Before
    public void Setup()
    {
      _model = new DatacentreModel();
      _provisioner = new Provisioner(_model);
      _app = new Application("App1");
      _model.addApp(_app);
      _model.addPool(_pool = Pool.createPool(_app));
      _overloaded = new Overloaded();
    }

    @Test
    public void notOverloaded()
    {
      assertFalse(_overloaded.testConditionEvent(_provisioner));
```

```
}

@Test
public void isOverloaded()
{
  _app.addRequests(new Request[]
        {
            new Request(1)
        });

  assertTrue(_overloaded.testConditionEvent(_provisioner));
}

@Test
public void serverAddedNotOverloaded()
{
  _pool.addServer(new Server(20));
  _app.addRequests(new Request[]
        {
            new Request(18)
        });

  assertFalse(_overloaded.testConditionEvent(_provisioner));
}

@Test
public void serverChangeOverloaded()
{
  Server server = new Server(20);
  _pool.addServer(new Server(20));
  _pool.addServer(server);
  _app.addRequests(new Request[]
        {
            new Request(28)
        });

  assertFalse(_overloaded.testConditionEvent(_provisioner));

  _pool.removeServer(server);
  assertTrue(_overloaded.testConditionEvent(_provisioner));
}

@Test
public void increaseRequests()
{
  _app.addRequests(new Request[]
        {
            new Request(20), new Request(20)
        });

  assertTrue(_overloaded.testConditionEvent(_provisioner));

  _pool.addServer(new Server(50));

  assertFalse(_overloaded.testConditionEvent(_provisioner));

  _app.addRequests(new Request[]
        {
            new Request(20), new Request(20), new Request(20)
        });

  assertTrue(_overloaded.testConditionEvent(_provisioner));
}

@Test
public void stillOverloaded()
{
  _app.addRequests(new Request[]
            {
                new Request(20), new Request(20)
            });

      assertTrue(_overloaded.testConditionEvent(_provisioner));

      _pool.addServer(new Server(30));

      assertTrue(_overloaded.testConditionEvent(_provisioner));
  }
}
```

B.3.3 PoolingTest.java

```
package tests;

import datacentre.manager.AndConditionPassRHS;
import datacentre.manager.CreatePoolForApp;
import datacentre.manager.ExcessPools;
import datacentre.manager.HavePools;
import datacentre.manager.ProficientPooling;
import datacentre.manager.Provisioner;
import datacentre.manager.RemoveExcessPool;
import datacentre.model.Application;
import datacentre.model.DatacentreModel;
import datacentre.model.Pool;
import org.junit.Before;
import org.junit.Test;
import trProgram.AndCondition;
import trProgram.Conditional;
import trProgram.IsTrue;
import trProgram.Nil;
import trProgram.NotCondition;
import trProgram.TRProgram;
import static org.junit.Assert.*;

public class PoolingTest
{

    private DatacentreModel _model;
    private TRProgram _provisioner;
    private ProficientPooling _proficientPooling;

    @Before
    public void Setup()
    {
      _model = new DatacentreModel();
      _provisioner = new Provisioner(_model);
      _proficientPooling = new ProficientPooling();
      _provisioner.addActionToBottom(new Nil(_proficientPooling));
      _provisioner.addActionToBottom(new CreatePoolForApp(
            new AndConditionPassRHS(
            new NotCondition(new ExcessPools()),
            new NotCondition(new HavePools()))));
      _provisioner.addActionToBottom(
            new RemoveExcessPool(new ExcessPools()));
    }

    @Test
    public void proficientPoolingEmpty()
    {
      _provisioner.singleRun();

      assertTrue(_provisioner.getTrueConditions()
            .contains(_proficientPooling));
    }

    @Test
```

```
public void proficientPoolingOneApp()
{
  _model.addApp(new Application("App1"));

  _provisioner.singleRun();

  assertFalse(_provisioner.getTrueConditions()
        .contains(_proficientPooling));
}

@Test
public void proficientPoolingOneApp2ndRun()
{
  _model.addApp(new Application("App1"));

  _provisioner.singleRun();
  _provisioner.singleRun();

  assertTrue(_provisioner.getTrueConditions()
        .contains(_proficientPooling));
}

@Test
public void badAppsBadPools()
{
  Application _app1;
  Application _app2;
  Application _app3;
  Application _app4;
  _model.addApp(_app1 = new Application("app1"));
  _model.addApp(_app2 = new Application("app2"));
  _model.addApp(_app3 = new Application("app3"));
  _model.addApp(_app4 = new Application("app4"));
  _model.addPool(Pool.createPool(_app2));
```

```
      _model.addPool(Pool.createPool(new Application("app5")));
      _model.addPool(Pool.createPool(_app4));
      _model.addPool(Pool.createPool(new Application("app6")));
      _model.addPool(Pool.createPool(new Application("app7")));

      for (int i = 0; i < 2; i++)
      {
          _provisioner.singleRun();
      }

      assertFalse(_provisioner.getTrueConditions()
            .contains(_proficientPooling));

      for (int i = 0; i < 20; i++)
      {
          _provisioner.singleRun();
      }

      assertTrue(_provisioner.getTrueConditions()
            .contains(_proficientPooling));
    }
}
```