University of Newcastle upon Tyne
School of Computer Science

Towards Certifiable Reconfigurable Real-time Mission Critical Software Systems

by
Richard Wilkinson

PhD Thesis

January 12, 2009

Abstract

This thesis makes a contribution towards the certification of reconfigurable real-time mission critical software systems. In highly reconfigurable software systems it is possible for a situation to arise where the system expends most or all of its resources on reconfiguring, and thus cannot provide sufficient resources to conduct intended computing functions. This anomaly has been termed "configuration thrashing" by the author due to its loose analogy to memory thrashing. If configuration thrashing is not eliminated, or at least minimised, then it is possible for circumstance to occur where reconfigurable systems cannot be certified due to potential failure to meet deadlines caused by configuration thrashing. The elimination of reconfiguration thrashing is a step towards certifiable dynamic reconfigurable systems capable of enforcing deadlines. The elimination of reconfiguration thrashing is necessary, though not sufficient, for this goal.

In order to restrict configuration thrashing it is necessary to understand the possibilities available within reconfigurable software. A VDM-SL model is presented to explore the options available for reconfigurable architectures, and has allowed many operators to be formally specified providing a much greater understanding of the tasks involved in reconfiguration.

The thesis demonstrates how model checkers can be used to check software processes for configuration thrashing using predefined CSP models, thus allowing system programmers to engineer configuration thrashing out of systems. However, model checkers are susceptible to state space explosion, particularly if models are large and / or complex, which may make the use of the model checkers impractical or even impossible for some systems. The thesis therefore also explores potential run-time solutions to configuration thrashing. These solutions allow developers to include additional logic / processes within their systems in order to eliminate configuration thrashing (without the use of model checkers). Several options are explored in-depth, from providing mechanisms for developers to choose when reconfiguration can / cannot occur, to a rule based solution. The exploration of the rule based solution explores issues such as rule expression, rule predictability, as well as potential core rules.

The two approaches taken within this thesis to eliminate, or at least restrict sufficiently, configuration thrashing form a basis which would allow for the certification of reconfigurable real-time mission critical software systems.

Acknowledgements

This thesis would never have been completed without the invaluable help of many individuals during the course of this study.

I would like to express my gratitude to my Supervisors at the University of Newcastle: Professor Tom Anderson for his encouragement and focus during the final stages of the writeup, and Dr Jim Armstrong for the encouragement and assistance given during the early work and for commenting upon aspects of the work even when no longer at the University of Newcastle.

Thanks are also due to Dr Steven Paynter of MBDA UK Limited whom during several meetings provided help and insight, particularly on technical matters relating to the FDR model checker. Thanks must also go to BAE SYSTEMS for the funding provided.

I am deeply grateful to my family, in particular my partner Helen, for their support, encourangement, and most of all patience during this period of study.

Contents

Chapter 1

Introduction

A reconfigurable system is one designed at the outset for changes in its structure, this may be hardware and / or software components, in order to adjust to environment changes. Reconfigurable systems have a high level of flexibility; allowing changes to occur much more quickly than in traditional non-reconfigurable systems.

Two types of reconfiguration can exist within reconfigurable systems: reconfigurable hardware and reconfigurable software. Reconfiguration can take place online or offline, though online reconfiguration offers the most potential benefit.

The avionics industry is investing heavily in Integrated Modular Systems (IMS), which is a movement towards a reconfigurable fault tolerant architecture in the avionics domain. This investment clearly shows the perceived value that the industry places on the benefits expected from reconfigurable systems.

In highly reconfigurable systems it is possible for a situation to arise where a system expends most or all of its resources on reconfiguring, and thus cannot provide sufficient resources to conduct intended computing functions. This anomaly has been termed "configuration thrashing" by the author due to its loose analogy to memory thrashing. This thesis addresses the problem of configuration thrashing and proposes strategies to eliminate, or at least restrict sufficiently.

The rest of this chapter is structured as follows. First Section 1.1 briefly introduces reconfigurable systems. Subsections 1.1.1 and 1.1.2 describe reconfigurable hardware and reconfigurable software respectively, followed by a brief comparison of reconfigurable hardware and software in subsection 1.1.3. Section 1.2 then introduces Integrated Modular Systems. Within section 1.3 configuration thrashing is discussed. Section 1.4 discusses the contribution this thesis makes and Section 1.5 presents the thesis structure.

1.1 Introduction to Reconfigurable Systems

Reconfigurable systems offer the ability to adapt hardware and / or software to meet changing requirements. Reconfiguration can take place online or offline. As discussed below, online reconfiguration offers the most potential benefit, but is also the most technically challenging.

1.1.1 Reconfigurable Hardware

Reconfigurable hardware devices, including Field-Programmable Gate Arrays (FPGAs), contain computational elements (often referred to as logic blocks) connected using (re)configurable routing resources. Custom digital circuits can be mapped to reconfigurable hardware devices by computing the logic functions in the logic blocks, and using the (re)configurable routing to connect the blocks together to form the desired circuit.

FPGAs and other reconfigurable computing devices have been shown to accelerate a variety of computing applications. For example, an implementation of the Serpent Block Cipher in the Xilinx Virtez XCV1000 shows a throughput increase by a factor of 18 compared to a Pentium Pro PC running at 200MHz [1].

In order to achieve performance benefits, yet support a wide range of applications, reconfigurable hardware devices are usually formed using a combination of reconfigurable logic blocks and a general-purpose microprocessor. The microprocessor performs the operations which cannot be done efficiently within the reconfigurable logic, such as data-dependent control and memory accesses.

Systems that are configured only at power-up (offline) are able to accelerate only as much of the program as will fit the programmable structures. Additional areas of a program might be accelerated by altering and reusing the reconfigurable hardware during program execution. This process is often known as Run-Time Reconfiguration (RTR) or online reconfiguration.

RTR has the benefit of allowing for the acceleration of a greater proportion of an application; however, it also introduces an overhead penalty incurred by (re)configuration which limits the amount of acceleration possible. Detailed information on reconfigurable hardware is contained in [2].

1.1.2 Reconfigurable Software

Reconfigurable software offers the ability to modify software systems either by reorganising or changing existing processes, adding new processes, or removing old processes.

Software systems that are configured only at start-up (offline) do not gain the benefits that online reconfiguration can offer. Offline reconfiguration allows for a system to be initialised in a number of different configurations, thus allowing the system to be optimised for the intended and foreseen life cycle. To reconfigure an offline reconfigurable system must shut down entirely and be reinitialised in the new configuration. Online reconfigurable systems can change configuration during operation; offering many potential benefits including: online software upgrades, adaptability, self-management, and increased fault-tolerance. Online reconfigurable systems are also referred to as dynamic reconfigurable systems.

Operating systems and programming languages have provided programmers with the ability to perform software changes at runtime for many years. However, such mechanisms have been said not to "...guarantee that a change will have the desired effect or maintain application integrity..." [3].

Dynamic reconfigurable software and specifically dynamic components have been identified as being "...challenging in terms of correctness, robustness, and efficiency..." [4]. To gain a better understanding of reconfigurable software many formal specification languages have been developed, though most of these are focussed on architecture specification or other specialist issues, rather than the reconfiguration actions themselves. A brief summary of fourteen specification approaches can be found in [5]. Chapter 4 of this thesis presents related work in which many formal specification languages are introduced.

1.1.3 Hardware and Software Similarities

Many similarities exist between the reconfigurability options available in hardware. Both hardware and software can be reconfigured offline or online and similar benefits can be gained by allowing online reconfiguration. Performance benefits and also increased adaptability are offered by online reconfiguration in both hardware and software. Both have difficulties when attempting to ensure the correctness of online reconfigurations. However, one difference is that hardware reconfiguration is likely to have an effect upon software and could potentially trigger subsequent reconfiguration within the software, but it is unlikely that

software reconfiguration will have an impact upon hardware or trigger any further hardware reconfiguration (unless specifically programmed to do so).

It is possible that hardware devices such as FPGAs could suffer from configuration thrashing, in the same way that software can. This can also mean that if software and hardware are made reconfigurable the issue of configuration thrashing could potentially be magnified.

1.2 Introduction to Integrated Modular Systems (IMS) for Avionics

Conventional aerospace systems are federated, with each major component hosted on separate hardware. This can be very costly, as each component is independently developed and validated. Validation is essential within aerospace systems, as they are complex real-time systems and the cost of a failure is likely be loss of life. Due to the potential for fatalities within the aerospace domain, all aerospace systems are regulated. In federated systems the software is generally tightly coupled to the hardware and is sensitive to small changes in either software or hardware.

The avionics industry is currently attempting to move to IMS, to allow for a fully dynamic network which can be used to meet mission targets successfully even in the event of system failures. IMS also aims to pool and share computing hardware, in order to reduce the overall cost of building systems. Cost is further reduced as less power, space and cooling is required. With IMS a number of software components can run on a shared processor and communicate via an operating system. Figure 1 shows the ARINC 653 [6] view of how this should be done.

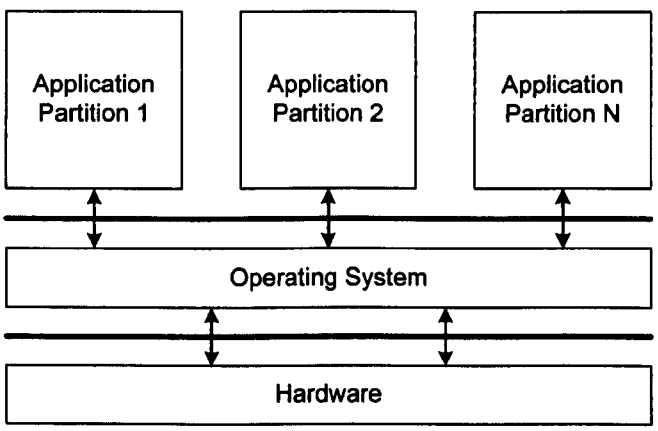

Figure 1: ARINC 653 IMS Module

As stated in [7] "... in order to benefit from the technology a safety case must be generated which can be maintained incrementally with system changes...". At present, if a small change is made in software or hardware, then recertification of entire components is required, and if the system is not federated into components, then the entire system would have to be recertified.

The fact that the avionics industry is investing so heavily in IMS is a clear indication that there is strong motivation for a loosely coupled, reconfigurable fault tolerant architecture in the avionics domain. In order to produce a dynamic reconfigurable system, suitable for that domain, there are many challenges that must be overcome. These challenges include:

1) Deadlines - real-time systems impose timing constraints. In hard real-time systems these deadlines must be enforced or catastrophic events may occur. In dynamic systems verification that such deadlines will be met is difficult as the system could be in any one of a very large range of possible configurations. There could be hundreds or even thousands of possible configurations in a given IMS system. There may also be a need for deadlines to be enforced during reconfiguration.

2) Validation / Certification – the validation and certification of real-time systems is a costly procedure. Complex reconfigurable systems will be difficult to certify and most likely even more expensive. However, modular safety cases could make certification of IMA systems easier and less costly.

1.3 Configuration Thrashing

As described, configuration thrashing is an anomaly which can occur in reconfigurable systems whereby a system expends most or all of its resources reconfiguring, and thus cannot provide enough resources to execute intended computing functions. To date no literature has been found that recognises or addresses configuration thrashing.

In order to give a concrete definition for configuration thrashing, the notion of a "configuration overlap" is introduced. A configuration overlap occurs when two subsequent reconfiguration requests are acted upon without a "sufficient interval" between them. The sufficient interval between reconfiguration actions should be sufficient to allow reconfiguration / initialisation to complete, as well as a minimum level of processing to occur (in the new configuration), and thus is an application dependent parameter.

Some (unusual) applications may have no required minimum level of processing; however, in most cases there will be a requirement for an

interval in which the system should conduct intended computing functions in order to justify the resource overhead of the reconfiguration. It is expected that in most cases the minimum processing period will at very least include an input and an output action, unless the internal state and process stack are to be maintained during reconfiguration. If the internal state and process stack are not maintained during reconfiguration, then the process will have made no progress if a read and write activity is not conducted.

Configuration thrashing occurs when one or more configuration overlaps occur. The number of configuration overlaps that can be tolerated in a given time period or in sequence is application dependent and possibly even mode dependent, as it is dependent on system deadlines. In some applications up to N consecutive overlaps could be tolerated, but no more. In others, the limit may be set at a maximum of M overlaps during any given time window of duration T.

If configuration thrashing is not eliminated, or at least minimised, then it is possible that no useful work might be achieved, deadlines may be missed, and certification may be impossible. The elimination or restriction of configuration thrashing provides a step towards certifiable dynamic reconfigurable systems. The importance of certifiable reconfigurable systems is not to be underestimated, as many real-time systems would benefit from reconfigurable functionality which must be certified for use, such as aerospace applications, or systems for power plants.

1.4 Thesis Contribution

This thesis makes a contribution towards the certification of reconfigurable real-time mission critical software systems, by investigating the configuration thrashing anomaly and providing methods of eliminating or restricting it sufficiently. This thesis considers certification as a crucial factor, as all aspects of the thesis are considered from an aerospace perspective; certification as well as safety are essential within this domain. However, the author has recognised that this research has applications in other domains and thus has tried to make all models and demonstrators as generic as possible. More specifically the contribution made in this thesis consists of:

1. the specification of a VDM-SL model providing the means to explore the characteristics of reconfigurable systems. This formal model has also allowed many relevant operators to be specified, thus providing a set of unambiguous reconfiguration operators.

2. a formal definition has been proposed for configuration thrashing and a formal modelling technique has been presented (using CSP and FDR) which is capable of detecting, and thus enabling the elimination of, configuration thrashing within software processes.

3. a range of run-time techniques for restricting reconfiguration and thus configuration thrashing has been explored. Since model checkers can suffer from state space explosion, a demonstrator has also been developed to further highlight that the run-time solutions can indeed restrict configuration thrashing sufficiently.

1.5 Thesis Structure

The remainder of this thesis is structured as follows: First, chapter 2 introduces a three-level model specified in VDM-SL which provides a basis for exploring the options available within reconfigurable systems. The types of options explored using the VDM-SL model include hardware reconfiguration, software reconfiguration, location awareness, and the effects of linkage upon reconfiguration as well as reconfiguration options. The operators outlined in the VDM-SL model form an extensible reconfiguration language. Chapter 3 outlines a definition of configuration thrashing which is then formally specified. Model checkers are introduced and CSP models are presented which enable model checkers to check processes for configuration thrashing. Discussions relating to the difficulties in applying the CSP configuration thrashing models to specific applications, such as terrain following aircraft which suffer from probabilistic requirements, are also presented.

Chapter 4 considers the different kinds of formalisms that could be used to model reconfigurable systems and thereby provide support for the identification of configuration thrashing. This chapter also presents related work on control techniques which could be used to constrain systems in order to ensure that configuration thrashing cannot occur. Chapter 5 explores potential run-time solutions to configuration thrashing. These solutions allow developers to include additional logic / processes into their systems in order to eliminate configuration thrashing (without the use of model checkers). A rule based solution is explored in depth and issues such as rule expression, rule predictability, and potential core rules are discussed. A demonstrator is also presented which shows that rules can indeed restrict reconfiguration sufficiently to eliminate configuration thrashing.

Chapter 6 reviews the effectiveness of both the models outlined for configuration thrashing and also the rule based software solution through

the use of a small case study. This chapter shows how useful both approaches can be to developers.

Chapter 7 presents proposals for future work, some of which are extensions to work presented within the thesis, and some of which fall outside the scope of the thesis. Chapter 8 presents the conclusions from the research performed.

Appendix A contains the full VDM-SL model described in chapter 2 of the thesis. The model is a three-level model and was developed to provide a basis for exploration of reconfigurable systems. Appendix B contains the full timed and un-timed CSP configuration thrashing models introduced in chapter 3 of this thesis. Appendix C presents possible processor, memory, OS, and storage requirements for processes, and appendix D presents the source code for the reconfigurable systems demonstrator. Appendix E contains both the full CSP configuration thrashing models and java source code used for the case study presented in chapter 6.

Chapter 2

Reconfigurable Systems

Reconfigurable systems offer the ability to adapt hardware and software to meet the changing requirements of a system. Reconfiguration can be static (when a system is off-line) or dynamic (at run-time), thus allowing the system to respond to changing requirements as it and its environment change.

Much research in reconfigurable systems is very focused and thus does not assess the options available, also initial research into reconfigurable systems has shown that many ambiguous terms are being used to describe the behaviour of reconfigurable systems, for example, many papers use terms such as "move" or "migrate" to describe the movement of a process between platforms. However, many questions are left unanswered by such terms, including: are process states migrated; are the communication links maintained; and are there any pre-conditions to the migration or move?

The rest of this chapter is structured as follows. First section 2.1 explores the options available for reconfigurable architectures. Section 2.2 introduces a suitable language in which to express the behaviour of reconfigurable systems. Section 2.2.1 introduces the VDM-SL model which has been used to formally define a candidate set of operators which make up the extensible reconfiguration language. Section 2.2.2 describes the individual operators in detail. Section 2.2.3 presents an ongoing debate as to whether formal models, including VDM-SL models, should be executable. Section 2.2.4 presents some conclusions relating to the operators defined.

2.1 Options for Reconfigurable Architectures

The exploration of possibilities available within reconfigurable architectures has shown that the following broad levels of reconfiguration are possible:

a) *Software (process) reconfiguration* – includes: process migration, process addition / deletion, thread spawning, dynamic linking and loading, state and stack synchronisation, as well as dynamic compilation and subsequent execution.
b) *Hardware reconfiguration* – includes: adding and removing hardware, reprogramming hardware (such as FPGAs), as well as changing hardware communication links.

c) *Mobility (hardware relocation)* – includes: the movement of hardware between physical locations.

As can be seen from the very broad levels of reconfiguration shown above, there are a large number of options available within reconfigurable systems. All of the options can be broken down further, for instance the removal of a process could simply remove the process from the processor its currently executing upon, or it could remove the process from the system entirely (including the executable file stored on non-volatile hardware). It could also remove it instantly or after a given time period. This gives a vast number of options within reconfigurable systems. All of the above reconfigurations could take place online or offline. Online reconfiguration offers specific benefits such as fault rectification, but presents a number of technical challenges (especially within real-time systems).

Most systems that support online reconfiguration will incorporate integrated systems such as "plug and play" and / or fault tolerant services. With these types of integrated systems, a reconfiguration in any one of the levels (outlined above) could trigger further reconfiguration. For example if hardware is removed and the software running on it is important, then the fault tolerant services will most likely trigger reconfiguration, to initialise the missing processes on different hardware. It is likely that such fault tolerant services would be turned off whilst upgrades take place, as they could potentially cause unwanted reconfigurations during upgrades.

As the options for reconfiguration are so vast the remainder of this chapter will explore software reconfiguration only, though the effects this has upon the underlying hardware will not be ignored.

2.2 Reconfigurable Operators

Research into reconfigurable systems has shown that a suitable language in which to express the behaviour of reconfigurable systems is lacking. Many ambiguous terms are used when describing reconfigurable systems, for example, the term "process migration" is often used without considering any of the following: is the process's state migrated; are the communication links maintained; and are there any pre-conditions to the migration?

A three-level model has been specified in VDM-SL to provide a basis for exploring the possibilities available within reconfigurable systems. The VDM-SL model has been built to allow an IMA type architecture to be manipulated using a set of well defined reconfigurable operators. However, the model is as generic as possible and can express almost any

reconfigurable architecture. The operators outlined form an extensible reconfiguration language.

2.2.1 The VDM-SL Model

The VDM-SL model consists of three levels: the process level, the hardware level, and the physical location level. All three levels are necessary to model a reconfigurable system accurately, as they are all required to analyse factors which affect resources. Most calculi capable of describing reconfigurable systems (such as the Pi-Calculus [8] and the Ambient Calculus [9]) fail to distinguish hardware location from physical location. It is important to separate hardware location from physical location not just because the resources required to achieve a task may alter with physical location, but also because mobile processes must be stopped and restarted when "in transit" between hardware locations, but will function continuously when in transit between physical locations.

The VDM-SL model has been built to allow an IMA architecture to be manipulated using a set of well-defined reconfigurable operators; however the model is as generic as possible and can express almost any reconfigurable architecture. To make the model generic, hardware links are individually modelled, even though IMA assumes a totally interconnected network. However, faults could occur in an IMA system which could cause the network to no longer be totally interconnected, and to model a scenario such as this, the individually modelled hardware links are necessary. The model also includes shared data areas, which would not be required in an IMA model, but may be required if a Real Time Network (RTN) [10] or similar approach to building reconfigurable systems were to be investigated.

The model is in essence a system state which is manipulated using a set of operators. The system state is shown in figure 1, along with the definitions for Hardware, Software and SW_to_HW_Map. Figure 1 does not include invariants, as they are not required here.

```
state System of
HardWare        : Hardware
SoftWare        : Software
Loc             : Locations
SW_HW_Map       : SW_to_HW_Map
HW_Loc_Map      : HW_to_Loc_Map

Hardware :: MAUs     : map MAU_ID to MAU
            Cards    : map Card_ID to Card
            Mappings : map Card_ID to MAU_ID
            Linkage  : map HW_Link_ID to HW_Link

Software :: Services       : map Service_ID to set of Global_Process_ID
            Processes      : map Global_Process_ID to Process
            SDs            : map Shared_Data_ID to Shared_Data
            Linkage        : map Link_ID to SW_Link
            SD_Linkage     : map Link_ID to Shared_Data_Link

SW_to_HW_Map :: Proc_to_Procrs : map Global_Process_ID to set of
Card_ID
                Proc_to_PMem   : map Global_Process_ID to set of
Card_ID
                Proc_to_NPMem  : map Global_Process_ID to set of
Card_ID
                SD_to_NPMem    : map Shared_Data_ID to set of Card_ID
```

Figure1: VDM-SL System State

The invariants placed over the model were kept as weak as possible to ensure that the maximum amount of possibilities for reconfiguration could be explored. It is envisaged that there could be many varying levels of "architectural constraints" placed over reconfigurable systems, which could restrain the possibilities for reconfiguration in many different ways. Developers may require differing levels of restriction in different projects.

Even with very weak architectural constraints (represented as system invariants), it was often difficult to ensure that operators did not violate the system invariants during reconfiguration. This is likely to become even more challenging if the number of invariants are increased, thus further restricting the reconfiguration. An example of this can be seen in the *ChangeLoadedProcID* operator within the model. This operator changes the *Global_Process_ID* of a process which is currently executing. In order for this to be done, it must change the *Global_Process_ID* in *Software.Processes*, as well as in the *SW_to_HW_Map* (amongst others). However, the system invariant states that any *Global_Process_ID* used in the *SW_to_HW_Map*, must exist in *Software.Processes* and this invariant will be broken if the *Software.Processes* is changed first, or if the *HW_to_SW_Map* is changed first. Within the model, this is overcome by wrapping the two actions in an atomic action to ensure that they both occur simultaneously (as a transaction).

All operators specified within the model are well-behaved, i.e. do not break any system invariants. However it is envisaged that further operators which are not well behaved may be required to model failures. For instance a failure could occur which effectively removes a card without un-initialising or de-allocating any of the processes allocated to it. In cases were system invariants are broken it is required that services will be available to

reconfigure the system to a valid state (a state where the invariants are no longer violated). Operators of this type have not been specified within the VDM-SL model, as they would only be useful if a fault tolerant service were to be specified to reconfigure the system to a valid state. It was not the aim of this research to investigate possibilities for fault tolerant services.

To simplify the VDM-SL model, processes have been modelled as single-threaded activities. The simplification was required as multithreaded applications could be spread over multiple processors. In cases where processes are allocated to sets of processors dynamically it is incredibly difficult to identify which threads need to communicate, and thus which hardware links would be required to support such communication. Furthermore in some applications processes spawn new threads dynamically making the analysis even more challenging. To analyse thread allocation in multithreaded applications, some form of graph theoretic approach would be required. The analysis of multithreaded application behaviour within reconfigurable systems is beyond the scope of this research; in fact it could form a PhD in its own right.

2.2.2 Software Reconfiguration Operators

Process reconfiguration can be split into two main types of operators: "*move*" and "*copy*". Other types of operator, such as process addition and process deletion, are not discussed in-depth in this chapter, as they are utilised within the *move* and *copy* type operators.

The following basic copy operators are outlined in the model:

- *CopyProc*
- *CopyProcWState*
- *CloneProc*
- *CopyProcWSWLinks*
- *CopyProcWStateAndSWLinks*
- *CloneProcWSWLinks*

The basic copy operators initialise a copy of a process (activity) on a selected set of hardware. A selected set of hardware must include a minimum of a processor, some persistent memory and some non-persistent memory. Within this thesis, this set of hardware will be referred to as a "computing platform". The initialisation is done from the executable of the original process. The operators which have 'State' in their name synchronise the state of the new process with the original process. The state represents the internal variables of a process. The operators which begin with 'Clone' synchronise the state and the stack of new process with the

original process. The stack represents the instruction stack (including the current position within the instruction stack).

Within this thesis, the set of processes with which a process communicates will be referred to as the "communicants" of the process. The operators with the 'SWLinks' suffix create software links to allow communication with the communicants of the original process. However, there is no guarantee that the hardware infrastructure will be able to route such communications.

An interesting point to note is that within the VDM-SL model it seemed necessary to allocate all processes a global unique identifier. If processes were not allocated global unique identifiers, then some form of location-dependent reference would be required in order to facilitate communicate between processes. Location-dependent references seem inadequate, since if a process were to be moved the reference for that process would change; thus all of the communicants of the (moved) process would have to be notified of the change. In systems where the hardware is half-duplex it may not be possible to notify processes of a change.

Within the VDM-SL model move operators were specified as low level primitives. In some systems move operators are implemented as copy operations followed by delete operations. The following basic move operators were outlined in the model:

- *MoveProcDelFirst*
- *MoveProcDelAfter*
- *MoveProcWState*
- *MoveProcWStateAndSync*

All of the basic move operators shown above, apart from the *MoveProcDelFirst* operator, initialise a copy of the chosen process (activity) on a computing platform and then remove the original process; this encompasses de-initialisation, de-allocation and then finally deletion (including from non-volatile hardware). The *MoveProcDelFirst* operator de-initialises the original process, and then reallocates the executable to a computing platform and initialises it, without removing the executable from non-volatile hardware. In an implementation of the *MoveProcDelFirst* operator, the executable would most likely be moved between non-volatile hardware. The operators which have 'State' in their name also synchronise the state of the new process with the original process, and the operators with 'Sync' in there name synchronise the stack in the new process with the original process.

As *Global_Process_ID*'s are unique within the system, it is necessary to allocate temporary identifiers to newly created processes when conducting most move operations. This is necessary as the original process is removed after the new process is created; as such the very last action that takes place in these move operations is to change the identifier of the moved process to the correct (original) identifier.

All of the operators described above have a pre-condition which states that the operation can only be attempted if there is a route between the present location of the original process and the new location specified for the moved or copied process. This is required as an operation of this type cannot take place if data cannot be sent between the chosen hardware nodes.

It is possible that if the network hardware is not totally interconnected, then once a copy or move has taken place, the newly created process can no longer communicate with the communicants of the original process (due to insufficient hardware linkage). A solution to this is to use proxies.

Proxies can be placed on hardware nodes to pass messages between processes. Three different types of proxies have been defined within the VDM-SL model, though more could be added. Figure 2 shows the three types of proxies which have been defined in the VDM-SL model.

```
Process = Activity | Proxy | Duplex_Proxy | Condensing_Proxy;

Proxy :: Source   : Global_Process_ID
         Target   : Global_Process_ID
         Activity : Activity

Duplex_Proxy :: Source   : Global_Process_ID
                Target1  : Global_Process_ID
                Target2  : Global_Process_ID
                Activity : Activity

Condensing_Proxy :: Source1 : Global_Process_ID
                    Source2 : Global_Process_ID
                    Target  : Global_Process_ID
                    Activity : Activity
```

Figure 2: Proxies Within VDM-SL Model

The three types of proxies defined have separate purposes. The standard proxy is a simple message relay proxy; it takes a message from its source process and passes it to its target process. The duplex proxy has one source and two targets; it receives messages and forwards them to two target processes. The duplex proxy could be particularly useful if a process or service (a set of processes) wishes to be duplicated for fault tolerance purposes. The condensing proxy has two sources and only one target; it receives messages from two separate processes and relays those messages to a single process. The condensing proxy could be implemented to conduct 'voting', thus only send one copy of a message even though it receives two,

or it could simply relay all messages. It is possible for proxies to be connected to other proxies, forming "chains" of proxies.

The following move and copy operators with automatic proxy generation are outlined in the model:

- *MoveProcDelFirstLP*
- *MoveProcDelAfterLP*
- *MoveProcWStateLP*
- *MoveProcWStateAndSyncLP*
- *CopyProcLP*
- *CopyProcWStateLP*
- *CloneProcLP*
- *CopyProcWSWLinksLP*
- *CopyProcWStateAndSWLinksLP*
- *CloneProcWSWLinksLP*

All of the above copy and move operators with automatic proxy generation behave as their parent operator (the operator with the name the same, but without 'LP' suffix), but also leave appropriate proxies to enable communication with the original communicants.

It is possible for operators to be specified which add constraints to the model (which will most likely trigger actions indirectly, though not immediately). An example of this would be an operator to keep two processes co-located. If an operator such as this were used, then a move operator (of any type) if called on either of the co-located processes would cause both processes to move. These types of operator have not been fully explored within the VDM-SL model, but some operators of this type may be useful within dynamic reconfigurable systems. It is envisaged that, in an implementation, services would have to be created to support such operators. These operators have not been fully explored in the VDM-SL model as VDM-SL does not support concurrency and as such a service to support such operators cannot be executed in parallel with reconfiguration operators to detect infringements on the constraints they introduce.

Assuming a faulty process would have the right to migrate itself and / or other processes within a system, then a faulty process could force process migration to occur continuously and thus cause "configuration thrashing". If configuration thrashing is not eliminated or at least minimised, then a reconfigurable system could expend most or all of its resources reconfiguring, and thus not provide sufficient resources to conduct intended computing functions.

There is also a requirement to prove that services provided for reconfiguration, such as fault tolerant services, cannot have errors of commission.

2.2.3 Executable vs. Non-executable Specifications

The VDM-SL model is an executable model. There is some dispute as to whether formal models should be executable. Hayes and Jones [11] present many arguments against this idea, which include the following:

- Executability limits the expressive power of a specification language and restricts the forms of specifications that can be used. Specifications should be phrased in terms of required properties of the system. They should not contain the algorithmic details necessary to make them directly executable.
- Though executable specifications permit early validation with respect to the requirements by executing individual test cases, proving general properties about a specification is much more powerful.
- Executable specifications can unnecessarily constrain the choice of possible implementations. Implementers can be tempted to follow the algorithmic structure of the specification although that may not be desirable. Executable specifications can produce particular results in cases where a more implicit specification may allow a number of different results.
- A specification language should be expressive enough to specify non-computable problems such as the halting problem. If it is not, one cannot use the single specification notation to cover both theoretical aspects of computing and practical ones.

Fuchs [12] argues for executable specifications by showing that non-executable specifications can be made executable on almost the same level of abstraction, without the introduction of new algorithms. Fuchs demonstrates that declarative specification languages allow a combination of expressiveness and executability.

Fuchs makes the following argument for not excluding executable specifications: "... all means applicable should be available to validate the specifications with respect to explicit and implicit requirements. Executable specifications can be crucial for this because they allow – in addition to formal reasoning about the specification – immediate validation by execution, and they provide users and developers with the touch-and-feel experience necessary to validate non-functional behaviour, e.g. user

interfaces. Excluding executability from specification languages means therefore depriving oneself of a powerful method of validation."

The VDM-SL model created has been made executable for the following reasons:

- Non-executable specification techniques can allow the specification of systems which are impossible to implement. Hayes and Jones note this as a positive point and an argument against executable specifications, however in the author's opinion it is a negative point when not working with theoretical aspects of computing. Within the VDM-SL model it was important to know that the operators specified could be implemented. Atomic actions have been used within the model to ensure system invariants are not violated during reconfiguration; however these atomic actions could be implemented.
- When making a specification executable, many significant implementation issues are drawn out. Making the model executable enabled the investigation of reconfiguration implementation issues without the need to implement a reconfigurable system.
- It is possible that ambiguous terms are being used to describe the behaviour of reconfigurable systems because system developers are not sure what is involved when reconfiguration takes place. For this reason, it is important for the reconfiguration operators to be detailed and understandable for system developers. An executable model is likely to be familiar for system developers and thus easier to understand.
- The model created provides a possible implementation method for the reconfigurable operators. However, there was no need for concern that implementers could be tempted to follow the algorithmic structure of the specification, as it only outlines a possible implementation, not an optimum one.
- As Fuchs as well as Hayes and Jones point out, an executable specification permitted early validation with respect to the requirements by executing individual test cases.

Some of the criticisms of executable models made by Hayes and Jones were found to be valid when creating the VDM-SL model, particularly the criticism that executable specifications lead to design decisions being made in the specification, which is too early in the development process.

It was sometimes difficult to make the specification executable without going into irrelevant (with respect to reconfiguration) algorithmic details. An example of this is the method used to generate unique identifiers. In an implementation a standard method would be used (a possible algorithm is

outlined in [13]), but to specify a particular standard algorithm in the VDM-SL model gave no benefit in the exploration of possibilities within reconfigurable systems. In circumstances such as this, a basic but not ideal specification was used and comments were placed in the model to highlight this.

2.2.4 Operator Conclusions

The VDM-SL model has shown that the number of options available within reconfigurable systems is much greater than anticipated. It has assisted in the exploration of possibilities for process reconfiguration, and has allowed many operators to be outlined formally. The operators form part of an unambiguous reconfiguration language for system developers to use in reconfigurable systems development.

The model has also given interesting insights into reconfigurable architectures. It has shown that it seems necessary to allocate all processes a global unique identifier, and to avoid reliance on (inadequate) location-dependent references. The model has also shown that even with minimal system invariants, some operations require atomic actions to ensure system invariants are not violating during reconfiguration.

Proxies are not commonly associated with reconfigurable systems, however the model has shown that they may have a valuable role to play in dynamic reconfigurable systems, when a totally interconnected network is not available. Proxies will not function adequately as (single threaded) activities, but instead should be multithreaded processes, as threads will most likely have to be spawned for each arriving message. This may mean that the resource requirements will be dependent upon the number of received messages and thus not predictable (without knowledge of the message rates).

This research has shown that the implementation of the operators specified in the VDM-SL model would be difficult, though not impossible. Certain operators may require OS support, for instance operators which synchronise processes instruction stacks may require OS support to write to such private memory areas. It should also be possible to verify the individual implementations of such operators.

Reconfiguration control is necessary for reconfigurable systems, particularly online reconfigurable systems, as processes must be constrained in order to ensure configuration thrashing cannot occur. Reconfiguration control is discussed further in chapter 5.

2.3 Summary

This chapter explores the options available for reconfigurable architectures and has identified three broad levels of reconfiguration: Software (process) reconfiguration, Hardware reconfiguration, and Mobility (hardware relocation). Each of these broad levels of reconfiguration present many options, and the exploration of the options available has highlighted that there are more options available within reconfigurable systems than anticipated.

Research into reconfigurable systems has shown that a suitable language in which to express the behaviour of reconfigurable systems is lacking. Many ambiguous terms are used when describing reconfigurable systems. A three-level model has been specified in VDM-SL to provide a basis for exploring the possibilities available within reconfigurable systems. The operators outlined in this model form an extensible reconfiguration language which is formally specified and as such unambiguous.

Many interesting insights have come from the exploration conducted into the options available for reconfigurable systems, including proxies, which are not commonly associated with reconfigurable systems, may have a valuable role to play in dynamic reconfigurable systems, when a totally interconnected network is not available. The exploration has also shown that certain operators may require OS support.

Chapter 3

Defining Configuration Thrashing

In highly reconfigurable systems it is possible for a situation to arise where a system cannot provide sufficient resources to conduct intended computing functions due to reconfiguration actions utilising required resources. This anomaly has been termed "configuration thrashing" by the author due to its loose analogy to memory thrashing.

This chapter outlines a definition of configuration thrashing which is then formally specified. Model checkers are introduced and CSP models capable of checking processes for configuration thrashing are presented. Discussions regarding configuration thrashing for specific applications are also presented.

This chapter is structured as follows. First section 3.1 introduces configuration thrashing and establishes a non-formal definition. Section 3.2 defines configuration thrashing in terms of sequences of events (traces) for processes. Section 3.3 introduces CSP and presents two models capable of checking if a process can "thrash"; a CSP model which is un-timed is introduced in Section 3.3.1 and a CSP model which is timed is introduced in section 3.3.2. Section 3.3.3 discusses the limitations of the CSP models presented. Section 3.4 discusses the difficulties found in applying the configuration thrashing models and presents an interesting discussion on probabilistic deadlines.

3.1 Configuration Thrashing Introduction

Configuration thrashing is in essence a lack of progress of intended computing functions (i.e. I/O processing) due to reconfiguration utilising required resources, thus causing deadlines to be missed. All (reconfigurable) systems require a certain level of responsiveness or progress to be made, thus implying the existence of deadlines. A requirement for progress implies that configuration thrashing can occur in all reconfigurable applications, even applications such as Microsoft Word or Microsoft Excel (provided they were reconfigurable).

It may be argued that deadlines in reconfigurable systems are missed due to non-reconfiguration functionality being inefficient, rather than reconfiguration actions utilising resources needed by non-reconfiguration functionality. However this argument does not hold, as in extreme cases reconfiguration could take place continuously, thus making it impossible for

non-reconfiguration functionality to make progress no matter how efficient it is. It is possible in some cases that improvements in non-reconfigurable processing could allow processes to meet there deadlines without reconfiguration alterations, however it will not be possible in all cases.

Given that configuration thrashing only occurs in systems with deadlines, configuration thrashing could be defined as occurring when a system misses a deadline due to a configuration change. However, a definition such as this is not adequate as there is often no way of showing that had a system not reconfigured it would have achieved its deadline. Proving a system would have met deadlines if certain re-configuration events had not occurred is difficult due to factors such as: possible hardware failures or external events (environmental stimuli) requiring mode changes.

In order to provide a practical definition of configuration thrashing, the notion of a configuration overlap is introduced. A configuration overlap occurs when two subsequent reconfiguration requests are acted upon without a "sufficient interval" between them. The sufficient interval between reconfiguration actions should allow reconfiguration / initialisation to complete, as well as a minimum level of processing to occur in the new configuration.

The minimum level of processing required in a given configuration is application dependent, though in most cases it is expected that it will include at least a read and write action otherwise progress would not have been made. This is discussed further in section 3.4. Depending upon the reconfiguration operator chosen, the minimum level of processing required may vary. Chapter 2 introduced a candidate set of reconfiguration operators. If a process were to be moved using the *MoveProcDelFirst* operator, then the minimum level of processing would always be the same, as the reconfigured process would always start in its initial state. However, if a process were to be moved using the *MoveProcWStateAndSync* operator, then the reconfigured process would initialise as it was before the reconfiguration, and thus the minimum level of processing required may vary.

Four possible reconfiguration scenarios are presented in Figure 1; a configuration overlap occurs in scenarios A, B and C, but not D (provided reconfiguration requests are acted upon immediately following receipt).

In scenario A the process does not fully initialise before a new reconfiguration request is made. Note that a request for a process to reconfigure before initialisation has completed can only be acted upon if the process is reconfigured by a third party. In most cases initialisation is an atomic action meaning that OS support may be required to interrupt

initialisation, otherwise the reconfiguration must be delayed until the initialisation has completed. In scenario B the process does not start processing before a new reconfiguration request is made. A third party is likely to be required to act upon a request whilst the process is in the ready queue. In scenario C, the minimum processing time has not elapsed before a new reconfiguration request is made. In this scenario the request could be acted upon by the process or a third party. In scenario D a configuration overlap has not occurred, as the reconfiguration / initialisation and minimum processing time have both occurred in the interval between reconfiguration request N and N+1.

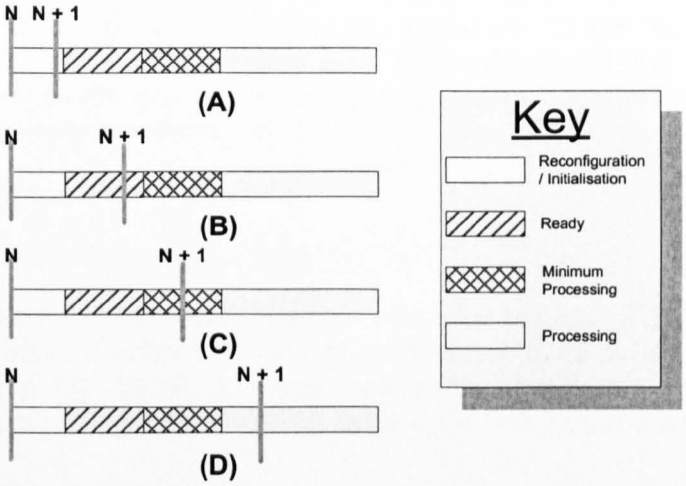

Figure 1: Overlap Scenarios

Given the notion of a configuration overlap, configuration thrashing can be defined as occurring when one or more configuration overlaps occur. The number of configuration overlaps that can be tolerated in a given time period or in a given sequence is application dependent and possibly even mode dependent. The worst case scenario is an infinite series of consecutive configuration overlaps, which will always be classified as configuration thrashing as progress cannot be made.

If configuration thrashing is not eliminated then it is possible for a situation to arise where a reconfigurable system cannot provide sufficient resources to conduct non-reconfigurable computing functions due to reconfiguration actions utilising required resources. The elimination of configuration thrashing is a step towards certifiable dynamic reconfigurable systems capable of meeting deadlines.

3.2 Trace Models of Configuration Thrashing

Configuration thrashing can be defined in terms of sequences of events (traces) for processes. A trace of the behaviour of a process is a finite sequence of symbols recording the events a process has engaged in up to some point in time. A trace is denoted as a sequence of events separated by commas and enclosed in angular brackets. For example the trace <*x*, *y*> consists of two events; *x* followed by *y*.

Before a process begins it is not known which of the possible traces will occur; the choice is dependent upon environmental factors beyond the control of the process. The complete set of all possible traces of a process *P* can be known in advance, this is defined as *traces(P)*. Some examples of traces for processes are shown below:

```
traces(STOP) = {<>}

traces(coin->STOP) = {<>, <coin>}
```

A configuration overlap occurs in a trace which has two *reconfigure* events occurring without the completion of a minimum level of processing between them. Given the following events: *reconfigure*, *begin_min_work*, and *end_min_work*, configuration overlaps occur in the following traces:

```
<reconfigure, reconfigure>

<reconfigure, begin_min_work, reconfigure>
```

A configuration overlap does not occur if the minimum level of processing is completed between *reconfigure* events, for example a configuration overlap does not occur in the following trace:

```
<reconfigure, begin_min_work, end_min_work, reconfigure>
```

If in an example, configuration thrashing is defined as two consecutive configuration overlaps for a process, the following traces would be examples of configuration thrashing:

```
<reconfigure, begin_min_work, reconfigure, begin_min_work,
reconfigure>

<reconfigure, reconfigure, reconfigure>

<reconfigure, begin_min_work, reconfigure, reconfigure>
```

However, based on the same definition the following trace would not be considered configuration thrashing:

```
<reconfigure, begin_min_work, reconfigure, begin_min_work,
end_min_work, reconfigure, reconfigure>
```

As the configuration thrashing definition varies from application to application, the above trace could be classified as configuration thrashing given a slightly different configuration thrashing definition. For instance if configuration thrashing was defined as a single configuration overlap the above trace would considered as configuration thrashing.

For configuration thrashing to be possible within a process there must exist at least one trace matching the configuration thrashing definition.

3.3 Communicating Sequential Processes (CSP)

CSP [14, 15] is a state-based behavioural notation developed for formally specifying sequential processes composed to run concurrently. CSP is used to specify concurrent processes, comprised of events on which process synchronisation can take place. CSP was one of the first process algebras (developed at the University of Oxford during the 1980s), and is one of the most widely used, along with Milner's Calculus of Communicating Systems (CCS)[16].

CSP represents a process as the set of sequences of its possible actions. Tool support is provided for CSP through FDR2. FDR2 directly supports three refinement models:

- **The traces model**: a process is represented by the set of finite sequences of actions it can perform.
- **The stable failures model**: a process is represented by its traces as above and also by its failures. A failure is a pair *(s,X)*, where s is a finite trace of the process (i.e., a trace from *traces(P)*) and *X* is a set of events it can refuse after *s*. The set of P's failures is given by *failures(P)*.
- **The failures/divergences model**: a process is represented by its failures as above, together with its divergences. A divergence is a finite trace during or after which the process can perform an infinite sequence of consecutive internal actions.

In general the traces model is used to check safety properties, the stable failures model is used to check deadlock freedom and the failures/divergences model is used to check livelock freedom.

3.3.1 Un-timed CSP Configuration Thrashing Model

This section introduces a CSP model capable of checking if a process can "thrash" that has been defined. The CSP model outlined includes several assumptions which have been made to simplify the model:

- The model is intended to check individual processes for configuration thrashing – this assumption has been made as it is logical to start with individual processes. However as discussed in section 3.3.3 this does lead to a limitation in the model.
- Processes within the model are single threaded – this assumption has been made to simplify the models produced (multithreaded models will be much larger and more cumbersome). However, multithreaded processes could be modelled and checked with some extensions to the model. An example of how threads can be modelled in CSP is shown in [17].
- Reconfiguration actions are not required to be detailed (i.e. a single action / operator can be used to model all reconfiguration types) – this assumption was made as there is no need to detail the reconfiguration actions in order to detect configuration thrashing, which is the aim of the model. It also minimises the alphabet of the system and reduces modelling complexities for developers.

The CSP model includes the following action types (alphabet): *reconfigure* which is used abstractly to represent any type of reconfiguration action / operator (chapter 2 introduced a candidate set of reconfiguration operators); *startup* which represents process initialisation; *doa* which represents an arbitrary internal processing action; *overlap* which signifies an overlap has occurred; *start_min_wk* which represent the beginning of the minimum processing period; *end_min_wk* which represents the end of the minimum processing period; and lastly *thrash* which represents a configuration thrashing occurrence.

The model consists of a monitor, a thrashing definition and an example process. Definition 3.3.1 shows the *MONITOR* which is used to specify how a reconfigurable process may behave, as well as identifying configuration overlap occurrences. As can be seen from the *MONITOR* definition, a reconfigurable process can first *startup* (initialise) or be reconfigured before initialisation. The *startup* event could be split into two events, one for the start and one for the end of the event, thus allowing reconfiguration to take place during initialisation. The *startup* event has not been split in this model as in most cases initialisation will be atomic, thus

requiring OS support to allow its interruption. A reconfiguration before or during initialisation would be a configuration overlap, as no useful work would have been completed in the new configuration. When a reconfigurable process has initialised, it can either do its minimum level of work and possibly additional processing, then reconfigure, or it can reconfigure before it has completed its minimum level of work. A reconfiguration before the minimum work has completed would be a configuration overlap. Note that within this model, the *reconfigure* action is used to signify both the reconfiguration request and action. Reconfiguration requests and actions are not modelled separately as they would occur consecutively. A time delay could exist between a request being made by one process and being received by another, however in CSP this is modelled as event synchronisation and thus there is no benefit in drawing a distinction between the reconfiguration request and action.

Definition 3.3.1

```
MONITOR = startup ->
            (start_min_wk -> reconfigure -> overlap ->
MONITOR
                [] start_min_wk -> end_min_wk -> reconfigure
-> MONITOR
                [] reconfigure -> overlap -> MONITOR)
          [] reconfigure -> overlap -> MONITOR
```

Configuration thrashing is defined within the CSP model using the *THRASH* process as shown in definition 3.3.2. This process allows configuration thrashing to be defined as a number of consecutive overlaps. The *THRASH* process works by taking variables for the maximum number of consecutive overlaps (*max*) and the number of overlaps remaining before configuration thrashing occurs (*x*). The variable *x* is normally initialised to *max* (signifying no overlaps have occurred). When a configuration overlap is detected *x* is decremented, and if the minimum level of work is completed then *x* is reset to *max*. When *x* reaches 0 configuration thrashing is detected, a *thrash* event is triggered, and the process stops.

The *THRASH* process does not contain a hard-coded variable for the number of configuration overlaps required, as this varies from application to application, and as such allows greater flexibility within the model.

Definition 3.3.2

```
THRASH(max, x) = if (x==0) then
                   thrash -> STOP
                 else
                   overlap -> THRASH(max, x-1)
                   [] end_min_wk -> THRASH(max,max)
```

Definition 3.3.3 shows an example process. This particular process is inherently capable of "thrashing" as it could engage an infinite sequence of *reconfigure* events. More complex processes can be checked using this model, as well as entire systems. Note that it is possible for a process capable of "thrashing" to exist as part of a system that is not capable of "thrashing" due to event synchronisation between concurrent processes eliminating certain traces from occurring.

Definition 3.3.3

```
PROCESS = startup ->
            (start_min_wk -> reconfigure -> PROCESS
               [] start_min_wk -> end_min_wk -> doa ->
               reconfigure ->       PROCESS)
               [] reconfigure -> PROCESS
```

In order to check processes against a configuration thrashing definition, additional processes are required. Definition 3.3.4 defines *SYSTEM* which ensures that the process being tested follows the structure of a reconfigurable process by the sharing its actions with the *MONITOR* (definition 3.3.1). This also allows configuration overlaps to be detected. *SYSTEM* hides non-essential actions, such as *startup*, *reconfigure* and *doa*. The hiding of non-essential actions allows trace refinement to be verified.

Definition 3.3.4

```
SYSTEM = (MONITOR ||{|startup,move,start_min_wk,end_min_wk|}||

(PROCESS\{doa}))\{startup,start_min_wk,reconfigure}
```

It was first thought that configuration thrashing could be detected using proposition 3.3.1. Note that the *THRASH* process used in this proposition does not include a *thrash* event, as shown in Definition 3.3.5. Many processes were checked for configuration thrashing using proposition 3.3.1, including the example process shown in definition 3.3.3, and all produced expected results. This gave increased confidence in the model and the definition of configuration thrashing. However, this proposition was found to be incorrect, as if the process being trace-refined could not "thrash" in all of the ways in which the *THRASH* process can, then *THRASH* does not trace-refine the process and thus proposition 3.3.1 gives a negative result, even if the process can "thrash". This highlights the point that model checkers cannot check if a property (or refinement) is specified correctly, or if a model is correct, thus it is possible for false positives or negatives to occur if a model or property is specified incorrectly.

Definition 3.3.5

```
THRASH(max, x) = if (x==0) then
                   STOP
                 else
                   overlap -> THRASH(max, x-1)
                   [] end_min_wk -> THRASH(max,max)
```

Proposition 3.3.1

```
assert SYSTEM [T THRASH(3,3)
```

An example of how an incorrect refinement assertion could give a false negative result can be seen when testing the example process shown in definition 3.3.6 using the refinement assertion presented in proposition 3.3.1. Definition 3.3.6 shows an example process which must "thrash", as it can only engage in events leading to overlaps. However, proposition 3.3.1 would give a (false) negative result for this process, as *THRASH* (definition 3.3.5) contains traces which *SYSTEM* (definition 3.3.4) does not. Some examples of traces that *traces(THRASH)* would have that *traces(SYSTEM)* would not are shown below:

```
<overlap, end_min_wk, overlap, overlap, overlap>

<end_min_wk, overlap, end_min_wk, overlap>

<overlap, end_min_wk, overlap, end_min_wk, overlap, overlap,
overlap>
```

Definition 3.3.6

```
PROCESS = startup -> start_min_wk -> reconfigure -> PROCESS
                [] reconfigure -> PROCESS
```

To produce the correct refinement assertion, the thrashing definition was extended to include a *thrash* action, as well as the *TEST* process. Process *TEST* (definition 3.3.7) defines configuration thrashing for the given scenario. In this particular definition configuration thrashing is defined as three consecutive overlaps. The *TEST* process also hides non-essential actions.

Definition 3.3.7

```
TEST = (SYSTEM |[{overlap, end_min_wk}]| THRASH(3,3))
          \{overlap,end_min_wk}
```

The correct refinement assertion (proposition 3.3.2) checks if *STOP* trace refines *TEST* (definition 3.3.7). As all events have been hidden apart from

thrash (the event that signifies a configuration thrashing occurrence), this assertion is true if configuration thrashing cannot occur and false if configuration thrashing is possible.

Proposition 3.3.2

```
assert STOP [T= TEST
```

Trace refinement is used in proposition 3.3.2, though stable failure refinement would give the same results. When conducting stable failure refinement, a process is represented by its traces and by its failures. A failure is a pair *(s,X)*, where *s* is a finite trace and *X* is a set of events it can refuse after *s*. All of the actions within *TEST* are hidden (internal) apart from the *thrash* action, thus if the process can "thrash" then in at least one of its traces it must have to accept a *thrash* action. However, *STOP* can refuse this action which would make the assertion false. As all of the actions are hidden apart from *thrash*, a process which cannot thrash would make the assertion true as it never refuses an action.

Failures/divergences refinement is not suitable, as a process which cannot "thrash" will produce an infinite sequence of consecutive internal actions (diverge), thus making the assertion false. A CSP divergence is a finite trace during or after which the process can perform an infinite sequence of consecutive internal actions. Failures/divergences refinement is not suitable due to the decision to hide all actions apart from the *trash* action within the model. An example process which could cause a divergence is shown in definition 3.3.8.

Definition 3.3.8

```
PROCESSNT = startup ->
            (start_min_wk -> reconfigure -> startup ->
start_min_wk ->
              end_min_wk -> doa -> reconfigure -> PROCESSNT
              [] start_min_wk -> end_min_wk -> doa ->
reconfigure ->
                 PROCESSNT)
            [] reconfigure -> startup -> start_min_wk ->
end_min_wk ->
              reconfigure -> PROCESSNT
```

3.3.2 Timed CSP Configuration Thrashing Model

Defining configuration thrashing using *THRASH* (definition 3.3.2) is adequate if configuration thrashing is to be defined in terms of consecutive configuration overlaps. However, it may be required that configuration

thrashing be defined as x overlaps in a given time period. CSP in its traditional form has no notion of time, though there are two distinct approaches to expressing time in CSP. The more elegant is to re-interpret the CSP language to record the exact time at which each event occurs. A trace thus consists of a series of time/event pairs, rather than just events. This theory of Timed CSP [18] adopts a dense, continuous model of time.

The alternative approach is a discrete model of time, which makes the drum-beat of time an explicit event. The interval between successive "beats" may be any finite duration. The drum-beat event representing the passage of time is conventionally named *tock* in CSP, as *tick* is a keyword in many tools including FDR2.

The discrete approach to modelling time was adopted to extend the un-timed CSP model (described in section 3.3.1), as although the continuous approach (as used in Timed CSP) is more elegant and corresponds to the standard way in which we think about time, the discrete approach offers the tool support needed for experimentation.

To extend the model described in section 3.3.1 to include time, the alphabet was extended to include the *tock* event. A *TOCKS* process (definition 3.3.9) was also added. This process is run in parallel with the other processes.

Definition 3.3.9

```
TOCKS = tock -> TOCKS
```

The only process not effected by the introduction of time to the model is the *MONITOR* process (definition 3.3.1), which remains unchanged. The *MONITOR* is unaffected, as there are no restrictions on how long actions should take in reconfigurable processes.

Configuration thrashing is defined in the timed CSP model using the *THRASHTIMED* process (definition 3.3.10 and 3.3.11). This allows configuration thrashing to be defined as a number of configuration overlaps in a given time period (a given number of *tock* events). If configuration thrashing is detected using this process, a *thrash* event is triggered.

To allow *THRASHTIMED* to detect configuration thrashing, an event history must be maintained. A novel approach to maintaining an event history is used within the model. This approach maintains a sequence of events only as long as is required to detect configuration thrashing. The number of events required in order to detect configuration thrashing varies, as all of the events which occur in the specified time period (*maxt tock* events) are required. Any number of *overlap* events could occur between

tock events. Figure 2 shows a basic situation where new events are added to the event history.

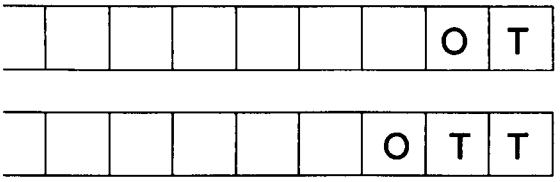

Figure 2: Event History Growth

The *THRASHTIMED* process maintains the correct sequence by using a *slidewindow* process to remove stale events from the event history. If a new *tock* event occurs whilst the event history contains *maxt tock* events (the specified time interval), the *slidewindow* process not only removes stale *tock* events, but also stale overlap events. Figure 3 shows a situation where a tock event (*T*) is added to the event history, when the event history already contains the maximum number of *tock* events (two) and thus stale events are removed.

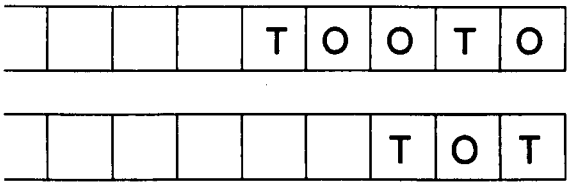

Figure 3: Event History Stale Removal

An example of the values maintained in the event history if *maxt* is set to three and one overlap event occurs after every two *tock* events is shown below:

```
<T>
<T,T>
<T,T,O>
<T,T,O,T>
<T,O,T,T>
<T,O,T,T,O>
<T,T,O,T>
```

The unique approach taken to maintaining the event history allows fresh history events to be maintained, whilst discarding stale events. This method essentially forms an extensible sliding window by which to check if configuration thrashing has occurred.

Definition 3.3.10

```
THRASHTIMED(<>, maxt, maxo) = overlap ->
THRASHTIMED(<>^<O>,maxt,maxo)
                  [] tock -> THRASHTIMED(<>^<T>,maxt,maxo)
```

Definition 3.3.11

```
THRASHTIMED(x, maxt, maxo) = if (numo(x) == maxo) then
                      thrash -> STOP
                    else
                      overlap -> THRASHTIMED(x^<O>,maxt,maxo)
                      [] tock -> if (numt(x) == maxt) then

THRASHTIMED(slidewindow(x)^<T>,maxt,maxo)
                         else
                           THRASHTIMED(x^<T>,maxt,maxo)
```

Definition 3.3.12 shows an example process. This process is capable of thrashing, provided the definition of thrashing is x overlaps or less in $x+1$ time intervals. As with the un-timed model more complex processes can be checked in the model as well as entire systems.

Definition 3.3.12

```
PROCESS = startup -> tock ->
            (start_min_wk -> tock -> reconfigure -> tock ->
PROCESS
                [] start_min_wk -> tock -> end_min_wk -> tock
                -> doa ->  tock -> reconfigure -> tock ->
                PROCESS)
              [] reconfigure -> tock -> PROCESS
```

The *SYSTEM* process (definition 3.3.13) has the same purpose as *SYSTEM* in the un-timed model; however it has been extended to ensure that the *TOCKS* process is synchronised upon.

Definition 3.3.13

```
SYSTEM = (MONITOR |[{startup,move,start_min_wk,end_min_wk}]|
            (PROCESS\{doa}))
            \{end_min_wk,startup,start_min_wk,move}
            |[{tock}]| TOCKS
```

The *TEST* process (definition 3.3.14) has the same purpose as *TEST* in the un-timed model. The *THRASHTIMED* process has replaced the *THRASH* process and *tock* events have both been shared and hidden. Definition 3.3.14 defines configuration thrashing as two overlaps in three time units.

Definition 3.3.14

```
TEST = (SYSTEM|[{overlap,
tock}]|THRASHTIMED(<>,2,3))\{tock,overlap}
```

Proposition 3.3.2 is used in the timed model, as well as the un-timed model. Trace refinement is still used, though as with the un-timed model, stable failure refinement could also be used. As discussed in section 3.3.1 failure/divergence refinement is not suitable for this model.

3.3.3 Limitations of CSP Configuration Thrashing Models

The CSP models presented within this chapter have been shown capable of detecting configuration thrashing and as such assist developers to engineer configuration thrashing out of their systems, but the models do have some limitations.

For example both CSP models only allow configuration thrashing to be detected in single processes and as such do not consider the reconfiguration of groups of processes. The author recognises that, particularly in distributed systems, complex interactions will exist between processes and as such developers are likely to consider reconfiguration as a step from one system layout (or blueprint) to another, which is likely to include many processes reconfiguring simultaneously or in a well defined sequence. The models produced can check if each individual process can "thrash" and as such be used to check entire systems (one process at a time), but this does not consider the fact that interactions between the groups of processes may make the processes that in theory can "thrash" not capable of configuration thrashing as the interacting processes may not be capable of producing the necessary stimuli to trigger the configuration thrashing.

Another limitation of the CSP models is that the *reconfigure* action / operator used in the models abstractly represents any type of reconfiguration within the model. As detailed above an assumption was made that no detail was required upon the reconfiguration action as this is not required in order to detect configuration thrashing. However, whilst modelling example systems it was found that as no information was available within the model as to the state of the processes during a reconfiguration action thus no reasoning could be made in relation to the reconfiguration action and any invariants that existed within the system. Although the model was not designed to review invariants it would be useful.

The final limitation to be highlighted for the models is one that exists because of the method in which model checkers such as FDR, the one used

within this thesis, function. Model checkers are susceptible to state space explosion [19]. This is particularly true of large complex models, which may make the use of the model checkers impractical or even impossible for some systems. As this is the case the models are limited in the size and complexity of processes which they can check.

Appendix B contains both the un-timed and timed CSP models capable of checking if a process can "thrash".

3.4 Difficulties Applying the Configuration Thrashing Model

The minimum level of processing required between two consecutive reconfiguration events is application dependent, as is the number of overlaps which must occur in a given time period or in a given sequence in order to be classified as "thrashing". Both of these may even be mode dependent.

It is expected that in most cases the minimum level of processing required between two consecutive reconfiguration events will include at least a read and a write action; otherwise progress would not have been made: if no read action is made the process will have no data to conduct processing upon, and if no write action is made it cannot store or provide other processes with the output of the processing.

The minimum level of processing required between reconfigurations is governed by deadlines. In general it is expected that processing deadlines will be more important that reconfiguration actions, thus reconfiguration should not interfere with deadlines. Deadlines can be divided into three types [20]:

- **Hard**: deadlines must not be missed. If not met considered fatal failure and may have disastrous consequences.
- **Soft**: deadlines can be missed. Missing soft deadlines is considered tolerable; there are no serious consequences. After the deadline has passed, delivery is still useful and thus required. The usefulness of delivery decreases over time.
- **Firm**: deadlines can be missed. No serious consequences. Late delivery is not required. If a firm deadline is missed the task is aborted.

In some real-time systems components may have probabilistic requirements. An example of a probabilistic requirement could be: component *A* must produce an output every 450ms at least 20% of the time or catastrophic

events may occur. With probabilistic requirements, the importance of a task finishing before its deadline will be dependent upon previous events. Using the example outlined above, it is possible for a situation to arise where the first 100 outputs are made within 450ms and thus the next output is not required to be within the 450ms deadline (it has either a soft or firm deadline depending upon the system requirements for delivery of late messages), as a late output will allow the system to maintain a greater than 20% ratio for on-time outputs, thus satisfying the requirement. However, if only 20 of the first 100 outputs had been produced within the 450ms deadline, then the next output would have a hard deadline, as a late output would reduce the percentage of outputs produced within 450ms to below 20% and thus the requirement would not be satisfied.

Probabilistic requirements can cause problems for system developers when trying to analyse system deadlines. In some cases developers make all deadlines over probabilistic requirements hard, in order to alleviate complications. This leads to over-engineering, but satisfies the requirements. However, in some cases over engineered requirements could conflict with normal (non over engineered) requirements unnecessarily.

Within most real-time systems probabilistic requirements are a symptom of their need to "synchronise" with the environment. The reason environment "synchronisation" is required is to avoid a situation where certain events can no longer be guaranteed to occur or no longer be guaranteed not to occur due to a lack of up to date environment information. For instance, a terrain following aircraft requires up to date terrain information, to ensure that it can avoid obstacles safely. If the aircraft becomes significantly out of step with its environment then it may not be possible to guarantee collision avoidance. Figure 4 shows an aircraft scanning its terrain (the environment) for potential obstacles. In this example it is imperative that the aircraft not miss two consecutive scans, otherwise it may not be able to guarantee collision avoidance. The diagram shows that at point 3 it would no longer be able to avoid the obstacle in its path. However a successful scan at either of the previous points would have enabled the aircraft to safely manoeuvre around the obstacle. A deadline for radar scanning in this example would be history dependent, as it is only a hard deadline if the previous deadline was missed.

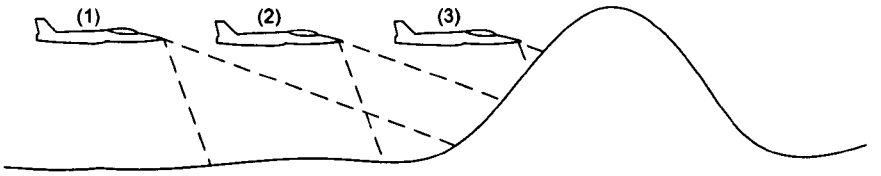

Figure 4: Terrain Scanning Diagram

As configuration thrashing is a lack of progress of intended computing functions (i.e. I/O processing) due to reconfiguration utilising required resources, thus causing deadlines to be missed, it is necessary for system developers to identify their deadlines in order to express and ensure that configuration thrashing cannot occur. However, deadlines are difficult for system designers and developers to draw from specifications. Often deadlines for systems are gathered from testing results. This can mean that deadlines are not 100% accurate and can also mean that systems are over-engineered to meet deadlines which could be relaxed.

Due to configuration thrashing being dependent upon system deadlines, it is impossible to give general answers to what the application dependent parts of the configuration thrashing definition should be. As discussed above it is expected that the minimum level of processing required between two consecutive reconfiguration events will include at least a read and a write action; otherwise progress would not have been made. Depending upon the reconfiguration action triggered, the minimum level required may vary, for instance if a process were to be moved using *MoveProcDelFirst* then the minimum level of processing will always be the same. However, if a process were to be moved using *MoveProcWStateAndSync*, then the reconfigured process would initialise as it was before the reconfiguration, thus the minimum level of processing required may vary. The process may not need a read and write action when the reconfiguration action used transfers the process stack, as the process will be initialised with the progress it had already made prior to the reconfiguration.

If a process is cyclic it is likely that the minimum level of processing would be a single cycle (though it could be broke down into smaller sections). Many applications are cyclic and thus a good starting point for system developers would be to make a cycle the minimum level of processing and then divide it into smaller sections if required.

As general advice cannot be provided for application dependent parts of the configuration thrashing definition, the simplistic terrain following radar

example shown in figure 4 is described in further detail, as well as methods to decide upon application dependent parts of the configuration thrashing definition.

Terrain following radar is an aerospace technology that allows a low flying aircraft to automatically maintain a constant distance above the ground, while flying at high speeds. The system works by periodically transmitting a radar signal downward and slightly forward (shown in figure 4). A computer computes the aircraft's height from the ground based on the signal's round-trip time and alters the aircraft's altitude in such a way as to keep a constant height above the ground just ahead, thus safely avoiding obstacles.

Terrain following radar is a necessity for high-speed low flying aircraft, since a human pilot cannot react quickly enough to changing terrain heights, and has a much larger probability of crashing into an unexpected mountainside than an automated system in the same circumstances.

As described a terrain following system periodically transmits a radar signal and conducts computations upon returned results. Thus the terrain following system (or subsystem) behaves in a cyclic fashion, meaning the minimum level of processing can be set to one cycle for the system. It is possible that this could be set to multiple or partial cycles, however without more exact information regarding the system, an exact decision on the minimum level of processing cannot be made.

As can be seen from the level of overlap on radar signals shown in figure 4, the number of consecutive overlaps which are required in order to ensure collision avoidance is two, thus the number of consecutive overlaps which must occur in a given sequence in order to be classified as configuration thrashing is two. As can be seen from this example, two consecutive overlaps does not necessarily mean that the aircraft will collide with something, though it is necessary to avoid two overlaps in order to guarantee that the aircraft will not collide with any obstacles (which may exist).

It should be noted that even an automated system utilising terrain following radar has a limited response time. Therefore, each system has a list of limitation in terms of the combination of maximal speed and minimal altitude allowed. The exact limitation figures change with radar type, with aircraft type and weight, and with the current meteorological conditions.

3.6 Summary

This chapter has introduced configuration thrashing, and highlighted how severe configuration thrashing can be. The worst case scenario described in this chapter shows that an infinite series of consecutive configuration overlaps can stop progress in any application. Configuration thrashing has also been explored using traces.

Two unique CSP models have been specified which enable developers to model their systems / processes and detect configuration thrashing potential. This allows developers to engineer configuration thrashing out of their systems. The first CSP model has no notion of time and allows developers to detect configuration thrashing when defined as a sequence of overlaps and the second model has a notion of time and allows configuration thrashing to be detected when defined as a number of overlaps in a given period of time. Limitations in relation to the CSP models have also been highlighted and discussed.

Interesting difficulties have been found when applying the CSP models to scenarios. In particular probabilistic requirements have been explored, where a deadline becomes a deadline only a percentage of the time, leading to configuration thrashing becoming dependent upon previous deadline achievement. This is quite a fascinating problem and although it has been solved for a particular scenario, it is difficult to provide general advice and as such this issue will cause developers problems when allowing reconfiguration within applications.

Chapter 4

Related Work

Within Chapter 3 a definition of configuration thrashing is formally specified. Model checkers are introduced and *CSP* models capable of checking processes for configuration thrashing are presented. Many formalisms could have been utilised to define configuration thrashing and check processes for configuration thrashing; *CSP* was chosen because of its tool support. Section 4.1 presents a brief overview of many "reconfigurable formalisms" which are capable of modelling configuration thrashing and in many cases, are capable of detecting configuration thrashing within a model and thus could have been used, instead of *CSP*, within the work presented in chapter 3.

As already stated to date no literature has been found that recognises or addresses configuration thrashing directly. However, this chapter explores some related work to investigate if configuration thrashing can occur within them and to put configuration thrashing into context next to similar problems found and / or addressed in related areas of work.

This chapter also presents relevant work on control techniques (section 4.3) which could potentially be used to constrain systems in order to ensure that configuration thrashing cannot occur. Chapter 5 builds on these control techniques and further explores potential run-time solutions to configuration thrashing; allowing developers to include additional logic / processes in their systems in order to eliminate configuration thrashing.

The rest of this chapter is structured as follows. Section 4.1 reviews various formalisms capable of modelling reconfigurable systems. First, early algebras are introduced in the form of *CCS* and *CSP*. Then more recent process algebras focussed upon mobility are described in the form of the *pi-calculus* and the *ambient calculus*. Lastly *Mobile Unity* is discussed, which is a formal notation designed for describing concurrent, distributed, and mobile computing systems.

Section 4.2 explores related work in the areas of *fault-tolerance*, *reflection*, *self modifying code* and *re-configurability* in general to put configuration thrashing in context with similar problems found in these related research topics.

Section 4.3 examines control techniques which could or in rare cases have been used to control reconfiguration. First *Law Governed Interaction* is

described, which provides a method of enforcing explicit coordination policies in a decentralised manner. Then the *Open Control Platform* is introduced, which is a software infrastructure for complex systems that coordinates distributed interactions and supports dynamic reconfiguration. Within this approach change application policies are used to allow changes to be made without violating reliability, safety, or consistency. Lastly, a *reconfiguration management* system is summarised. This system requires that all affected nodes and their neighbours be in a quiescent state before any reconfiguration occurs. While a node is in a quiescent state, it is prohibited from initiating communication. This ensures that nodes directly affected by a change will not receive communication during the course of a change.

4.1 Reconfigurable Formalisms

Reconfigurable systems offer the ability to adapt hardware and / or software to meet changing requirements. Many formalisms exist which are capable of modelling reconfigurable systems. Some example formalisms are introduced within this section. First two early process algebras are introduced; these are known as *CCS* and *CSP*. Other process algebras exist, but *CCS* and *CSP* were chosen as they are the most widely used. Both have proved themselves to be invaluable tools in the formal specification and verification of concurrent communicating systems. They are also both capable of modelling configuration thrashing and can both be used to check that process definitions cannot suffer from configuration thrashing. However, these process algebras are limited in that they cannot represent process creation, process deletion, or changes in process connectivity.

More recent process algebras focussed upon mobility are then introduced in the form of the *pi-calculus* and the *ambient calculus*. The *pi-calculus* is essentially an extension of *CCS* which adds the ability to pass channel names as parameters along communication channels. This allows receiving processes to communicate via channels they previously had no knowledge of. Within the *ambient calculus* mobility is modelled using the concept of ambients. An ambient is informally defined as a bounded place where computation can occur. Both the *pi-calculus* and the *ambient calculus* could be used to model configuration thrashing.

Mobile Unity is a formal notation designed for describing concurrent, distributed, and mobile computing systems. *Mobile Unity* has semantics in *Category Theory* and thus is different to the process algebras, but is equally capable of allowing a configuration thrashing definition to be made.

4.1.1 CCS

The *Calculus of Communicating Systems (CCS)* [16] is a process calculus developed by Robin Milner in the early 1980s. This formal language includes primitives for describing parallel composition, choice operators and scope restriction. The expressions of the language are interpreted as labelled transition systems.

CCS includes a notion of bisimulation which Robin Milner refers to as "a kind of invariant holding between a pair of dynamic systems" [16]. Bisimulation provides a technique to prove two systems equivalent in terms of behaviour with respect to the actions that can be performed.

Tool support is available for *CCS* in the form of the *Concurrency WorkBench (CWB)*. By using this tool, the specification of a concurrent system can be analysed. The *CWB* is capable of displaying a simulation of a concurrent system specified in *CCS*, searching for deadlock states, testing for equality between two specifications, and determining if a system satisfies specified logical properties (e.g., safety or liveness).

CCS has no primitives / operators for mobility of processes, and has no method of altering process connectivity. *CCS* has no notion of time, though time can be added to a *CCS* model in a number of ways; one example of how time can be added to *CCS* is contained in [21].

4.1.2 CSP

CSP was developed by Tony Hoare at the University of Oxford during the 1980s, and is one of the most widely used process algebras. *CSP* [14, 15] is a state based behavioural notation developed for formally specifying sequential processes composed to run concurrently. Within *CSP* processes are comprised of events on which process synchronisation can take place.

Tool support is provided for *CSP* through *FDR2*. *FDR2* allows for refinement checking, determinism checking, as well as looking for deadlocks and divergences. As discussed in section 3.3, *FDR2* supports three refinement models: the traces model, the stable failures model, and the failures / divergences model.

CSP is similar to *CCS* in that it is very low level and has no primitives / operators for mobility of processes or any dynamic creation of connections. *CSP* has no notion of time, though time could be added to a *CSP* model to allow for real-time issues to be explored. A theory of *Timed CSP* [18] adds time to *CSP* by re-interpreting the *CSP* language to record the exact time at

which each event occurs. A trace thus consists of a series of time / event pairs, rather than just events.

Although *CSP* and *CCS* do not have native support for reconfiguration, they are included in this related work section, as operators for reconfiguration can be added. Reconfiguration operators can be represented in *CSP* through events. Reconfiguration events can't create new processes or delete processes, but they can clearly show that reconfiguration has occurred or is in the process of occurring. A simple example of how to represent reconfiguration within *CSP* would be to simply have an action named something like *reconfigure*.

4.1.3 Pi-Calculus

The *pi-calculus* [8, 22] was developed by Robin Milner as an algebra to enable communicating and mobile systems to be reasoned about (in a rigorous manner). The *pi-calculus* is built upon *CCS*; it adds the ability to pass channel names as parameters along channels. This allows receiving processes to communicate via channels they previously had no knowledge of.

In the *pi-calculus*, the definition of bisimulation equivalence may be based on either the reduction semantics or on the labelled transition semantics. There are (at least) three different ways of defining labelled bisimulation equivalence in the *pi-calculus*: early, late and open bisimilarity. This stems from the fact that the *pi-calculus* is a value-passing process calculus.

The *pi-calculus* provides a framework for the representation, simulation, analysis and verification of mobile communication systems. Processes in the *pi-calculus* interact with one another by sending and receiving messages in a synchronous manner. Note that the calculus is non-deterministic; when several options are available the interaction that occurs is chosen on a completely random basis.

The *pi-calculus* does not explicitly model locations. However, physical locations can be represented using processes; the location of a process being modelled as a link between a process and a special "location" process. For locations to be modelled in such a way, it would be expected that each process would link to exactly one "location" process. A change in location would consist of breaking a link to a "location" process and creating a link to another.

As stated above, the pi-calculus allows a process to pass a channel (a communication path) to another process, and then that process can then

communicate via that channel to a (possibly) previously unknown process. An example of this is shown in Figure 1. Figure 1(a) shows three processes in which A can not directly communicate with C. However, if B passes the channel BC to A, then A can communicate with C via BC as shown in figure 1(b). It is not made clear (in the *pi-calculus*) if this is a similar situation to IP addresses or phone numbers (i.e. total interconnection is required at the hardware level) and so they communicate directly, or if the communication goes via B without the knowledge of B.

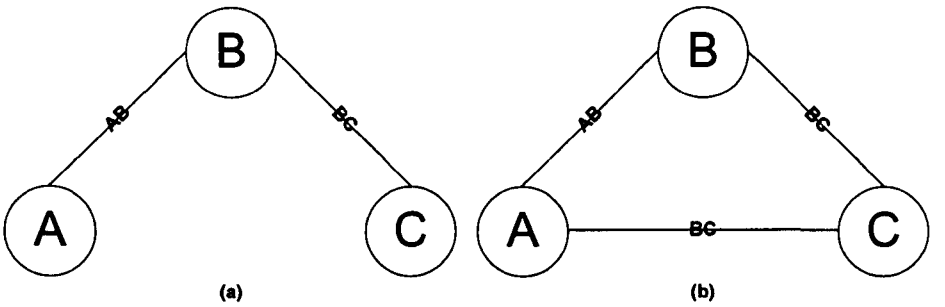

Figure 1: Pi-Calculus Example

There is some tool support for the pi-calculus. The *Mobility WorkBench (MWB)* is an automated tool for manipulating and analyzing mobile concurrent systems (those with evolving connectivity structures) described in the *pi-calculus*. The *pi-calculus* has no notion of timing, thus real-time issues have not been addressed within the *pi-calculus*.

4.1.4 Ambient Calculus

The *ambient calculus* [9, 23] is a process calculus developed by Luca Cardelli and Andrew Gordon to describe and analyse concurrent systems involving mobility. Within the ambient calculus mobility is modelled using the concept of ambients. An ambient is informally defined as a bounded place where computation can occur. Various informal interpretations have been given for the formal concept of an "ambient", for example ambients could include: a web page (bounded by a file), a Unix file system (bounded within a physical volume), or even a laptop (bounded by its case and data ports). Within the ambient calculus locations are represented using ambients, though ambients do not always represent locations.

Ambients can be nested within other ambients forming hierarchies. The *ambient calculus* adds the concept of capabilities to ambients to make it possible to model limited access to ambients. In particular, the ambient calculus supports *in*, *out* and *open* capabilities. The *in* capability instructs the ambient to enter a sibling ambient. The *out* capability instructs the

ambient to leave its named parent ambient. The *open* capability dissolves the boundary of an ambient.

Within the *ambient calculus* it is not enough for one ambient to simply know in which ambient another process resides in order to facilitate communication. The ambient must also know the 'route' – the hierarchical nesting of ambients.

The *ambient calculus* has a rich variety of operators, though no tool support has been found thus far. The ambient calculus has no notion of timing, thus real-time issues have not been addressed within it.

4.1.5 Mobile Unity

Mobile Unity [24-26] is a formal notation designed for describing concurrent, distributed, and mobile computing systems. Mobile Unity separates computation and coordination. Connectors in *Mobile Unity* are expressed in a program design language which has semantics in *Category Theory* [27].

The concept of a connector in *Mobile Unity* is used to express complex relationships between system components, thus facilitating the separation of coordination from computation. M. Wermelinger et al. [24] argue that the separation of coordination from computation "is especially important in mobile computing due to the transient nature of the interconnections that may exist between system components". The seperation of coordiation and computation that occurs in *Mobile Unity* provides a means for components to continue to function independently of the communication context in which they find themselves. However, the author would argue that this seperation is not especially important as most components behave differently with different stimuli, and different stimuli are likely to occur with different communication contexts, thus they are not completely decoupled.

Locations within *Mobile Unity* are modelled implicitly. Each component of a design is assigned a position. *Mobile Unity* allows a distinction to be made between programs which control their own motion and programs which are moved by the environment. This is done by declaring the location attribute as local or external, respectively. The method of expression of locations in *Mobile Unity* allows fine grained mobility to be expressed.

No tool support has been found for *Mobile Unity*. *Mobile Unity* has no notion of timing, thus real-time issues have not been addressed.

4.2 Reconfigurable Systems and Configuration Thrashing

Many types of system can reconfigure and most can suffer from problems similar in some way to configuration thrashing. This section explores related work in the areas of *fault-tolerance*, *reflection*, *self modifying code* and *re-configurability* both in general and in an effort to put configuration thrashing in context.

Each piece of related work is described and similar issues to configuration thrashing identified or addressed are explored. Any potential solutions to similar issues outlined are reviewed in terms of safety, formalisms used, and real-time applicability. In some cases pieces of related work can suffer from configuration thrashing and this has not been explored or recognised within the original research. In such cases this is documented.

4.2.1 Fault Tolerance

An important non-functional requirement that is often demanded of a system is fault-tolerance. Fault-tolerance demands that the system functions correctly, even in the presence of failures, regardless of the type. The concept of fault-tolerant computing has existed since at least the 1960's [28].

An early approach to providing software fault tolerance can be found in the recovery block scheme [29, 30]. Within this scheme logical blocks of code are separated and a framework is put around them. This framework first of all establishes a recovery point, then executes the code and then conducts an acceptance test to see if the output is acceptable (i.e. not faulty). In the event that the block of code fails a backward recovery takes place, i.e. the system moves back to the recovery point. Alternative blocks of code are then tried if available which must provide the same functionality as the first block (or primary block). Any number of standby spares can be provided for each block of code. This represents the basis of the recovery block scheme and is normally represented using the syntax shown in figure 1.

```
Ensure <acceptance test>
   By <primary module>
Else by <alternative module 1>
Else by <alternative module 2>
                .
                .
                .
Else by <alternative module n>
        Else error
```

Figure1: Recovery Block Scheme

If all of the modules fail then this is regarded as a failure of the recovery block and an exception will be signalled. Recovery blocks can be nested and thus one recovery block can form part of a module of an enclosing recovery block. Where recovery blocks are nested, if an exception is raised from a failure of an inner recovery block, recovery will take place and an alternative module of the outer recovery block will be executed.

Every time a fault occurs within recovery blocks a reconfiguration (of sorts) occurs; the system state is reset and some alternative code is then tried. This reconfiguration affects the progress made within the application as a backward recovery is undertaken and could lead to deadlines being missed. Since configuration thrashing is in raw terms a lack of progress of intended computing functions due to reconfiguration utilising required resources, thus causing deadlines to be missed, then a recovery block system can suffer from configuration thrashing.

As with most fault-tolerant research the priority for the research is recovering from a fault and there is no consideration made as to the meeting of deadlines. This is understandable; if the error is great then progress may be hindered unless a reconfiguration occurs and as such there may be no other option. However, in many cases the fault may be able to be tolerated for a period before reconfiguration which could lead to deadlines being met. Recovery block research has not considered real-time aspects and thus reconfiguration thrashing has not been addressed within the research.

The research on recovery blocks has been expanded upon and a concept of the Coordinated Atomic Action [31] (or CA Action) has been developed. This enhancement focuses upon providing a unified scheme for coordinating complex concurrent activities and supporting error recovery between multiple interacting objects in a distributed object-oriented system. This research can suffer a similar issue to configuration thrashing in much the same way that recovery blocks can. I.e. when faults the system still has to roll back, but instead of reviewing only single processes this research also reviews how multiple interacting processes can roll back in synch if an error occurs.

Another traditional software fault tolerant technique is N-version programming [32]. N-version programming works by providing N-versions of a program (N being greater than 1) which have been independently developed to a common specification and comparing their results by some form of replication check. A majority vote is then undertaken and erroneous (presumed faulty) results can be eliminated and the (presumed correct) results generated by the majority vote can be passed on. Within N-version programming the system remains static even when a fault occurs and although overheads are increased in terms of processing time by means of introducing N-versions and a voting algorithm deadlines should not be missed through the use of N-version programming. Systems using this method cannot suffer from configuration thrashing (at least not because of the fault tolerant method used).

More recent fault tolerant research has focussed upon the use of off the shelf applications in distributed environments. S Porcarelli et al [33] present a proposed architecture to deal with dynamic resource management for real-time dependability-critical distributed systems capable of coping with fault tolerance and scalability issues. This research builds upon the Light-weight Infrastructure for Reconfiguring Applications (Lira) and provides a decision making process to allow management as to how best to conduct / allocate a reconfiguration request. This research presents some very novel and interesting issues, but at present does not adequately address real-time issues. The proposed architecture does allow reconfiguration and thus processes within the architecture can suffer a lack of progress of intended computing functions due to reconfiguration utilising required resources causing deadlines to be missed. In theory the decision makers in this architecture could attempt to address configuration thrashing, but no reference to stopping certain reconfiguration requests or information relating to how deadlines would be enforced are discussed within papers found to date on this research. The concerns introduced relating to real-time issues were principally focussed upon the timeliness of the decision making process. No formal modelling has been undertaken within this research.

J. Fraga et al [34] present a component model for building distributed applications with fault-tolerant requirements and incorporates QoS requirements. Within this research a component model is build on top of CORBA and fault detection agents are used to detect faults. Once a fault is detected a change can be made. Using the QoS requirements given by components, the Adaptive Fault Tolerant (AFT) manager can change the configuration of the system when it detects QoS requirements are not being met. This research introduces a novel approach to attempting to ensure components can meet deadlines through QoS requirements. However, reconfiguring a system to avoid missing a deadline could make deadline

harder to achieve and this approach only seems to reconfigure once deadlines have already been missed, thus in a safety critical system this may be too late. This approach can suffer from configuration thrashing as no restrictions are made on reconfiguration in any way and this has not been considered as part of the original research.

4.2.2 Reflection

Reflection is the process by which a process / system can observe and modify its own behaviour. To facilitate the ability to observe itself a reflective system incorporates structures representing aspects of itself. The programming paradigm driven by reflection is call reflective programming. Uses of reflection include: maintaining performance statistics, debugging, decision making, self-optimisation, self-modification, and self-activation. Reflection is used in many modern programming / scripting languages such as: C#, Java, Perl, PHP, Curl and Python.

Applying reflection technology to middleware design has become an extensive research topic. Reflection offers more flexibility and adaptability to middleware systems. In [35] G. Coulson et al use a reflective component model as basis for constructing configurable and reconfigurable CORBA ORBs. Within this research it was recognised that the reflective component model inherently supported flexibility and reconfigurability, but on its own was too powerful and could easily lead to chaos. Component framework based structuring was introduced to impose domain specific constraints and semantics on the reconfiguration process. The research presented by G. Coulson et al addresses the following fundamental issues for effective reconfiguration management: i) to constrain the scope and effect of reconfiguration, ii) to separate concerns between reconfiguration operations and core middleware functionality, and iii) to maintain integrity in the face of dynamic change. It is possible within this research for the reconfiguration to cause a lack of progress of intended computing functions due to reconfiguration utilising required resources and thus can suffer from configuration thrashing. G. Coulson et al have not addressed this issue as their principle focus is upon the construction of a flexible reconfigurable middleware and as such this was most likely deemed as beyond the scope of the research, though the authors did recognise that constraints were needed for reconfiguration but not ones to reduce or eradicate configuration thrashing.

Research is presented in [36] that builds middleware (named RECOM) at the meta-level; they treat the binding between the specific client and the server as a self-representation of middleware. The self-representation completely reflects all aspects of the implementation of middleware. This

makes RECOM highly flexible. No research found to date reviews issues concerning timing or resource management in RECOM and as such configuration thrashing is possible.

DynamicTAO [37] has been developed as part of a research project conducted at the University of Illinios. DynamicTAO is a CORBA compliant reflective ORB that supports runtime reconfiguration. This type of implementation is a more typical one than that used for RECOM but does address issues such as security and safety for dynamic reconfiguration. DynamicTAO delegates resource management to components that can be dynamically loaded. It employs the Dynamic Soft Real-Time Scheduler (DSRT) [38], which runs as a user-level process in operating systems like Linux or Windows. The DSRT uses the system's low level real-time API to provide QoS guarantees to applications with real-time requirements and allows applications to specify QoS parameters in terms of CPU, memory and communication. It can therefore control the resource allocation to the quality desired by the application. However, the DRST provides no mechanism for reviewing the effects of reconfiguration and in fact the QoS requirements are monitored in a feedback loop fashion so if the QoS requirements are not being met, it may reconfigure the system itself to allow these to be met. DynamicTAO can suffer from configuration thrashing and no research found to date recognises this as an issue.

Similar research to that conducted on DynamicTAO is presented in [39] and presents a QoS framework implementation for OpenORB [40]. The OpenORB research describes a framework for supporting resource adaption by providing first-class representation of activities and generic interfaces for inspecting and controlling the resources allocated to activities. The OpenORB QoS framework research aims to represent the tasks and resources thus ensuring that if a given task requires x resource(s) to ensure a certain level of QoS then it is provided. However, this does not take into account that as well as a lack of resource QoS requirements could be broken by a high level of reconfiguration. Thus, as with the DynamicTAO research, the system can suffer from configuration thrashing.

Some general research concerning the benefits of using reflection is outlined in [41]. This research builds a framework to support dynamic adaption and aims to compare this to other reconfigurable techniques. This research clearly highlights the benefits of using reflection in that dynamic adaption can be achieved independently of the application's domain and also that the extension to add adaption functionality does not necessarily require changing the static class structure of the application. However, this research also highlights the disadvantage of a performance decrease. This and other general research on reflective programming does not look at its application to real-time systems and as such do not look at issues related to

configuration thrashing. The framework outlined in [41] can suffer from configuration thrashing, but this research paper does not address this.

4.2.3 Self-Modifying Code

Self-Modifying Code (SMC) is broadly referred to as any code that loads, generates or alters its own instructions at runtime. SMC can be used to improve performance in applications [42, 43] through runtime code generation. Dynamic code optimization can provide improved performance [44] or minimize code size for systems with a limited memory [45]. SMC can also enable dynamic code obfuscation [46] to support code protection.

SMC has been described as "a better strategy for realizing long-lived autonomous software systems than static code, regardless of how well it was validated and tested" [47]. This is mainly based on the idea that a self-modifying system can adapt to new situations better than static systems. A common technique applied to adapt programs is genetic programming [48, 49] in which programs are modeled as genetic material.

Research into run time code generation, such as the work presented in [42], make use of invariants and values that cannot be exploited at compile time, yielding what should be superior code. However, the cost of generating code at runtime can often be prohibitive. In general this type of research takes in code of one type, often a scripting language, and further optimises it; as such the system does not reconfigure and thus cannot suffer from reconfiguration thrashing in a traditional sense. However, within run time code generation it is possible for the reconfiguration in the code, that takes place during the code generation, to in itself cause a lack of progress of intended computing functions thus creating a very similar problem to configuration thrashing. Most of the research found to date attempt to tackle this problem by reducing the time taken to generate the code or by arguing the tradeoffs in terms of the time gained by using the optimised code. L Hornof et al [43] focus upon safety aspects of run-time code generation by proving the generated code meets the requirements of the original code, but does not address the issues surrounding the use of run-time code generation in real-time systems.

Dynamic code optimization is very similar to run time code generation, though what is referred to as optimization, as appose to generation, generally attempts to improve the performance of an instruction stream as it executes on a processor rather than generating new software from a given source. V. Bala et al [44] present a system named Dynamo; using this they show that in many cases even statically optimised native binaries can be accelerated. Dynamo is intended to provide a client-side performance

delivery mechanism that allows computer system vendors to provide some degree of machine-specific performance without the independent software vendor's involvement. This type of native-to-native runtime optimisation is conducted in a novel manner and its complete transparency offers many benefits. However, there are cases in which the optimisation can add to the execution time. As with run time code generation, dynamic code optimisation systems do not reconfigure and hence cannot suffer from configuration thrashing in a traditional sense. However, it is possible for the optimisation (reconfiguration) that takes place during the code optimisation, to itself cause a lack of progress of intended computing functions causing a similar problem to configuration thrashing. To date none of the research found focuses upon this issue, but instead argue that the rare cases in which execution times are decreased are minimised and are therefore acceptable to increase performance significantly in other processes / systems.

Using self modifying code for obfuscation is an unusual application of self modifying code, but a very effective one. Most reverse engineering techniques start by disassembling and then uses program analysis to recover high level semantic information. However the approach presented in [46] complicates this by changing the program code repeatedly during code execution thus thwarting disassembly. Templates are used to allow code to be altered back to the true executable code just in time to be executed. This technique can only be used on static code as mutating already changing source code would become impossible to track and ensure that the templates could allow the code to be correctly altered back ready for execution. It is possible for the continual changes in the source code to cause a lack of progress of intended computing functions due to code modifications utilising required resources and as such can suffer from configuration thrashing. However, this has not been addressed within any research found to date and in fact the slowing of code execution times is expected within this research as this is required to prevent reverse engineering.

An interesting technique applied to adapt programs using self modifying code is genetic programming [48, 49] in which programs are modeled as genetic material. The underlying theme for this type of research is that nature has developed a mechanism both for continuous operations and evolution and maybe this same method can solve the complexity that exists in the design, implementation, standardization and deployment of modern computing applications. Within the research presented in [48, 49] *fraglets* are introduced representing fragments of distributed computation. Fraglets have a strong formal methods tie and offer a lot of flexibility. Genetic programming allows for a highly dynamic system, though no research found to date ensures a safe system or ensures timing constraints. Genetically inspired modifications could lead to a lack of progress of intended computing functions due to code modifications utilising required resources

ergo this type of system can suffer from configuration thrashing, but no research found to date recognises or addresses such an issue.

Very little research found addresses any verification of correctness of self-modifying code. SMC is extremely difficult to reason about. Most existing formal verification techniques assume that code memory is fixed and immutable. H. Cia et el [50] have developed a technique for modular verification of general von-Neumann machine code with runtime code manipulation. This research is one of only a handful of pieces of research on verification of SMC. However, this research does not consider timing and as such cannot be used to verify if a piece of SMC can or cannot suffer from configuration thrashing, though it may be possible to extend the formal verification technique presented.

4.2.4 General Re-configurability

Reconfiguration is desirable as a run-time mechanism in most modern computing systems as it allows hardware and software upgrades in response to technological advancements, environmental changes, or alterations in requirements during system operation.

A lot of research into general reconfiguration mechanisms focuses upon the correctness of the system in relation to invariants placed on the system. These invariants are often checked after reconfiguration has occurred or in some cases throughout the reconfiguration process. S. S. Kulkarni et al [51] present one such piece of research in which they address the "lack of systematic methods to ensure the correctness of dynamic adaption" through the use of transitional-invariant lattice, which is based upon the concept of proof lattice, to verify correctness. This novel approach provides a formal method of using proof tools to show that invariants have been met throughout a reconfiguration and even coping with changing requirements (thus invariants) during a reconfiguration. It addresses safety through proving system invariants hold throughout reconfiguration, however this research does not address any timing issues and as such does not address any issues similar to or relating to configuration thrashing.

J. Zhang et al [52] present another approach to ensuring system's are correct (or "safe") during software change in their research. The method used to ensure that adaptations are safe with respect to system consistency takes into consideration dependency analysis for target components, specifically determining viable sequences of adaptive actions and those states in which an adaptive solution can be applied safely. This research is focussed upon the application of this method to changing external conditions in a wireless multicast video application. Invariants are used to specify dependency

relationships among components enabling the ability to determine which components are affected during a given change. A central management approach is taken in an attempt to provide optimisations where multiple change options are available and a rollback mechanism is provided in case an error or failure occurs. This research uses what would appear to be a very sound formal proof technique to ensure that changes are safe, however does have a few issues in that all invariants are specified by developers and as such if these are specified incorrectly then the method is useless. As with the research by S. S. Kulkarni et al, this research does not address any timing issues and does not address any issues similar to or relating to configuration thrashing even though the application considered should include timing concerns.

Research presented by S.K. Shrivastava et al [53] introduces architectural support for dynamic reconfiguration of large scale distributed systems. The approach presented in this research is based upon techniques from the area of workflow management. Workflows are pieces of rule based management software that co-ordinate and monitor execution of tasks. A task model has been developed that is expressive enough to temporal dependencies between tasks. To ensure that dynamic reconfiguration can occur safely several restrictions are placed on the system as to when reconfiguration can occur. These rules ensure that a task cannot be changed unless it is in a wait state and input / output alternatives cannot be added, removed or modified unless the tasks are in given states. Formal methods have not really been fully applied to this research though a rigorous approach has been taken. This research considers timing dependencies only and does not extensively explore deadline analysis. A system built using this architecture could suffer from configuration thrashing as the reconfiguration could still cause a lack of progress of intended computing functions.

Architecture Description Languages (ADLs) is a similar type of research to workflows. J. Magee et al [54] present research on dynamic software using an ADL called Darwin in which they explore dynamic features and illustrate some of the possibilities and problems in supporting constrained and unconstrained structural evolution. This research outlines an operational semantics in the *pi-calculus*. As it is focussed upon expressing and representing architectural designs, including components and their interactions, no timing issues have been explored and as such even though systems designed in this way can clearly suffer from configuration thrashing, it is not addressed. As no ordering or time information is available in ADLs then reconfiguration specifications are simply made of designs of the system and no consideration is made as to when a reconfiguration can occur or what state the system is in when reconfiguration occurs and also no consideration can really be made as to whether it is safe to reconfigure. Research into workflows seems more

suited to this type of analysis given the ordering of tasks that is available, but further research would need to be conducted in this area.

Quality of Service (QoS) is a more recent research topic. Quality of service generally refers to the ability to provide different priority to different applications, users, or data flows, or to guarantee a certain level of performance. A. Tesanovic et al [55, 56] present a novel approach to providing QoS in real-time systems by employing feedback-based scheduling methods. The feedback-based system uses a controller to calculate a deadline miss ratio and as such alters the system to ensure the ratio is kept within bounds. The research is used for performance-critical real-time systems and not hard real-time systems as by its very definition the miss ratio calculations shows that deadlines are missed and no bounds are placed on the system to control the length of time this can occur for which could be unsafe. This research does recognise that systems should be timely and that missing deadlines is undesirable, and as such does review a similar problem to configuration thrashing. However, this research views a short term deadline miss as no issue provided that the ratio is improved in the long term which is not acceptable for true real-time systems and thus does not solve configuration thrashing. Also as the system reconfigures when the ratio is detected to be outside of acceptable bounds, then this in theory could make configuration thrashing worse as no consequence for ongoing deadlines or the lack of progress of intended computing functions that will be caused by the reconfiguration. Such an approach could very easily reconfigure a component back and forth as neither configuration is quite meeting its QoS miss ratio which will only worsen the problem, although it is possible that in such a scenario nothing can truly be done to stop deadlines from being missed.

4.3 Reconfiguration Control

Reconfiguration control is necessary for reconfigurable systems, particularly online reconfigurable systems, as processes must be constrained in order to ensure configuration thrashing cannot occur. As stated in [57] "[w]ithout change management, risks introduced by runtime modifications may outweigh those associated with shutting down and restarting a system". The control techniques presented within this section could or in rare cases have been used to control reconfiguration.

4.3.1 Law Governed Interaction

It is argued in [58] that "... the great promise of architectural models has not been fulfilled so far, due to a gap between the model and the system it purports to describe ... this gap is best bridged if the model is not just stated, but is enforced". Law Governed Interaction (LGI) [58-63] has been designed as a coordination and control mechanism for heterogeneous distributed systems which can be used to enforce architectural principles.

LGI provides a method of enforcing explicit coordination policies in a decentralised manor. The law of the system is expressed as an explicit collection of rules relating to the structure of the system, the process of its evolution, as well as the evolution of the law itself. As this implies, the law of the system is not and should not be immutable, as this would be overly restrictive and not allow for evolving systems.

A whole range of principles can be enforced using LGI, including access control mechanisms, distributed coordination mechanisms, dynamic reconfiguration mechanisms, and various security mechanisms. A particular example of an architectural principle being enforced is highlighted in [63], where LGI is applied to the publish-subscribe paradigm to alleviate the "dark side" it has through the coordination of communication between agents. The dark side of the publish-subscribe paradigm comes from its decoupled communication which may complicate the system using it, making it less predictable, more brittle, and less safe.

LGI can be used to control reconfiguration by adding reconfiguration primitives into the law of the system. In [62] reconfiguration primitives are added into the law of the system, and a token ring example is presented to show the addition and removal of agents from the system. Changing the programming of a given agent without shutting down the system is achieved in [62] by having a controller for each agent absorb the requests arriving to it while the update of its programming is in progress (buffering is used).

LGI appears unsuitable (in its present form) for real-time systems, as it provides no method of ensuring that the system deadlines are met. It is stated in [58] that "[f]or an architectural principal to be defined into the law of an e-system it must be enforceable and the enforcement must be reasonably efficient". However, "reasonably efficient" is not good enough for real-time systems, and since any number of LGI rules can be placed in the law of the system, and they could be of any complexity, it is very difficult to carry out worst case analysis on the time taken to parse, interpret and apply LGI rule sets.

4.3.2 The Open Control Platform

The Open Control Platform (OCP) [64] is a software infrastructure for complex systems that coordinates distributed interactions and supports dynamic reconfiguration. Its primary goals are to: "... accommodate changing application requirements, incorporate new technology, interoperate in heterogeneous environments, and maintain viability in changing environments".

The OCP was specifically designed for complex dynamic systems such as aircraft, power systems, and telecommunication networks. The OCP is an entire platform that supports dynamic reconfiguration, thus its not just a reconfiguration control technique.

It is recognised in [64] that configuration management (control) is required "to ensure that the configurations are valid and consistent with the overall system requirements". The OCP takes an architecture oriented approach to reconfiguration management, building on work by Oreizy et al [57].

In this approach strategies referred to as change application policies are used to enable changes to be made without violating reliability, safety, or consistency. Policies within this approach are maintained separately from application specific behaviour, facilitating the ability to change policies independently of functional behaviour. A policy could state something along the lines of: to replace a specific component, the new component must be online for a given period of time before the old component is removed. This policy would allow the new component to synchronise with the old component.

The Oreizy et al approach distinguishes between two types of change, the first being changes in systems requirements, and the second being changes to system implementation that do not alter requirements. The first type, changes in system requirements, are not handled within the approach, as "[i]t is unrealistic to assume that any preconceived measures for maintaining system integrity would support this type of unpredictable and unrestricted change". The author agrees with this in principle; however, certain requirement changes could be foreseen and thus could be handled. It is not inconceivable for developers to anticipate certain requirement changes.

The OCP developers are extending the approach presented in [57] to real-time, mission critical applications, such as unmanned aerial vehicles. No technical information has been found to date that explicitly documents how the approach is extended to cope with real-time issues, though numerous articles do state that it is occurring.

4.3.3 Dynamic Change Management

Kramer and Magee [65] present a structural based approach to runtime change. The reconfiguration management system within this approach interfaces between the functional view of application programming and the structural view of the system configuration. Changes are specified in terms of system structure only. The system is modified by a series of transactions. These change transactions are derived by management from the change specification.

The reconfiguration management system requires that all affected nodes and their neighbours be in a quiescent state before the change is made, as well as during the change execution. This ensures that nodes directly affected by a change will not receive requests during the course of a change. This eliminates the need for buffering during reconfiguration. However, it is possible for a node to be connected to every other node in a system and thus when that first node requires reconfiguration the entire system must enter a quiescent state. This presents issues for real-time systems.

The management system is responsible for making decisions regarding application change. This is done based on a limited model of the application. This approach does not presently address real-time issues. However, it is likely that this approach would not be suitable for real-time applications because, as stated above, when changes are initiated portions of the system have to move to a quiescent state, thus potentially large portions of the system, if not the entire system, will make no progress while reconfiguration takes place.

4.4 Summary

This chapter has presented a brief overview of related work in the form of reconfigurable formalisms which are capable of modelling reconfigurable systems, as well as capable of detecting configuration thrashing within a model. Any of these formalisms could have been used within work presented in chapter 3; CSP was chosen because of its tool support. As this chapter has simply presented a brief overview it has not gone into detail as to how each individual formalism could have been used to detect configuration thrashing.

No related work has been found to date that specifically looks specifically at the issue of configuration thrashing, but this chapter has presented related work in the areas of *fault-tolerance*, *reflection*, *self modifying code* and general *re-configurability* in an effort to put configuration thrashing in context next to similar problems found in these areas of work. Many pieces

of similar work can suffer from configuration thrashing, but very few actually recognise that this is indeed an issue. The few that do recognise timing issues as a problem have reviewed this in terms of QoS and utilise feedback based algorithms which are not adequate for hard real-time systems. Also very few pieces of related work have a solid formal underpinning and the few that do, such as that presented by S. S. Kulkarni et al [51], focus upon safety through proving system invariants hold throughout reconfiguration, however this and other similar research does not address timing issues and as such does not and cannot address any issues similar to or relating to configuration thrashing.

This chapter also presents related work on control techniques which could be used to ensure configuration thrashing cannot occur. Chapter 5 further builds on these control techniques and explores potential run-time solutions to configuration thrashing. Several options are explored within Chapter 5, including providing mechanisms for developers to choose when reconfiguration can / cannot occur, and a rule based solution.

Chapter 5

Exploration of Potential Run-time Configuration Thrashing Solutions

Chapter 3 presents a formal definition of configuration thrashing along with timed and un-timed CSP models capable of checking processes for configuration thrashing (using FDR). These models can be used to ensure that configuration thrashing is engineered out of systems.

Some engineers may view producing models of their systems, in order to engineer configuration thrashing out of them, as excessively time consuming and also may lack the skills to produce such models accurately. Proving that thrashing cannot occur in an inaccurate model will not provide any guarantees for the system. Model checkers (such as FDR) are also susceptible to state space explosion [19]. This is particularly true of large complex models, which may make the use of the model checkers impractical or even impossible for some systems.

This chapter explores potential run-time solutions to configuration thrashing. These solutions allow developers to include additional logic / processes in their systems in order to eliminate configuration thrashing. Several options are explored in-depth, from providing mechanisms for developers to choose when reconfiguration can / cannot occur to a rule based solution. The exploration of the rule based solution investigates issues such as rule expression, rule predictability, as well as potential core rules.

The rest of this chapter is structured as follows. First Section 5.1 describes the options available for reconfiguration control. Section 5.2 then explores the options available when using rule sets to control reconfiguration. Within section 5.2.6 rule expression and rule predictability are discussed. Section 5.2.7 then describes the core rules which could be used within most, if not all, reconfigurable systems to eliminate configuration thrashing. Section 5.2.8 introduces a demonstrator which has been developed to highlight that the solution outlined using rule sets can sufficiently restrict reconfiguration and thus eliminate configuration thrashing. Section 5.3 explores the possibilities available to provide developers with mechanisms to control reconfiguration, and section 5.3.1 reviews potential guidance for developers using mechanisms to control reconfiguration.

5.1 Reconfiguration Control

There are several options available to ensure configuration thrashing cannot occur at run-time without the use of modelling techniques. The options available for reconfiguration control range from allowing developers to decide when reconfiguration is appropriate, through to providing a reconfiguration control process (a reconfiguration controller) to decide when reconfiguration can and cannot occur.

One method of allowing developers to choose when reconfiguration can / cannot occur could be facilitated by providing engineers with the ability to declare windows of opportunity within source code for processes to reconfigure. This could be further developed to allow developers to specify the reconfiguration operator(s) which can be used during the period the window of opportunity is open. For example, a developer could open a window of opportunity that enables process replication, but not process migration. Chapter 2 introduced a candidate set of reconfiguration operators. Giving developers control over reconfiguration on a process by process basis is a novel approach, provided that guidance can be provided to them on how to do it. Without guidance, developers may not know when reconfiguration should take place or, more importantly, when it is "safe" for reconfiguration to take place. Developers may be tempted to develop more static systems than necessary, thus not gaining the full benefits that reconfiguration can provide. Also, allowing developers to control reconfiguration introduces the likelihood of human error.

To reduce the likelihood of developers developing more static systems than necessary, and thus not gaining the full benefits that reconfiguration can provide, whilst still allowing developers to choose when reconfiguration can and cannot occur, it is possible to assume that reconfiguration can occur at all times apart from when the developer declares that reconfiguration should not occur. This would allow for the protection of certain actions, such as the closure of files, which cannot be just abandoned at any point in order for reconfiguration to take place. This approach does not give the programmer complete control over the reconfiguration scheme - it merely places constraints on the solutions a systems architect might adopt later. It should be easy for developers to define "critical sections" using this approach and thus protect certain actions, but it is difficult for developers to decide when reconfiguration should be denied to eliminate configuration thrashing. To eliminate the unknowns it is possible that developers may declare much bigger sections than required to not allow reconfiguration and thus produce more static systems than required.

A more dynamic approach to reconfiguration control is to allow a reconfiguration controller to decide when reconfiguration can and cannot

occur. This type of approach allows system developers to focus on core development without concern for reconfiguration issues.

Traditionally logic within control processes is predefined in source code; however, benefits such as quicker upgrades and run-time alterations could be provided if the logic in a reconfiguration controller were to be specified in a set of rules. Rule sets would most likely have a standard core set of rules which could then be extended for specific applications / systems. It is possible in some cases that additional rules would be required for specific new components upon their addition to a system; these new rules could be added at run-time provided they do not contradict existing rules. Dynamic rule sets introduces many potential problems which are discussed further in section 5.2.5.

A method of runtime rule enforcement is then required and could be achieved through either centralised or decentralised means. Computational resources would be required for rule checking and enforcement which would not be required if developers took control of reconfiguration. However, if computational resources are kept to a minimum it is believed this approach would provide benefits over and above the computational resources required. The benefits provided by this approach are wide ranging and include controlling configuration thrashing by not allowing reconfiguration to occur when the next reconfiguration action would constitute configuration thrashing. Malfunctioning reconfigurable processes can also be controlled by specifying rules which the processes are designed to follow. For example, if a process is only designed to run on a certain processor type then a rule could be added to the system to ensure the process never ends up on a different processor type. This approach also provides the benefit of relieving developers of the task of deciding when it is "safe" for reconfiguration to occur.

Certain similarities can be drawn between the options available for reconfiguration control and the options available for scheduling. Within scheduling some researchers believe that processes should be coded to give up control of resources when appropriate and thus are never forced to release control of resources. This is similar to the method outlined for reconfiguration control, where developers can be given the power to choose when reconfiguration can or cannot occur. In both scheduling and reconfiguration control this method can introduce uncertainty for the programmer which detracts / distracts from the core development tasks. Within scheduling the other option is to use an algorithm or protocol to control which processes get what share of processing resources. Scheduling algorithms are discussed in-depth in [20]. This approach is very similar to the use of a reconfiguration controller to control reconfiguration. In both

scheduling and reconfiguration control this method utilises increased levels of processing resource.

Hybrid approaches of the two main reconfiguration control options outlined above are possible, but in general do not offer any further benefits. A hybrid approach would include logic which takes up computational resource and also require developers to state where reconfiguration should or should not occur, which still leads to uncertainty and potential for human errors.

5.2 Reconfiguration Control using Rule Sets

Reconfiguration control using rule sets introduces many options; the two primary options are how to apply rule checking and rule scope. Rule checking concerns where rule validation takes place and rule scope concerns where individual rules are valid. Both rule checking and rule scope can be centralised (global) or decentralised (local). Figure 1 illustrates the effects these two options can have upon a system when applied in either local or global scope.

	Local Rule Checking (decentralised)	**Global Rule Checking** (centralised)
Local Scope (decentralised)	Local subsystem rules, local enforcement	Local subsystem rules, central enforcement mechanism
Global Scope (centralised)	System wide rules, local enforcement mechanisms	System wide rules, enforced by a central mechanism

Figure 1: Possibilities for Reconfiguration Control Using Rule Sets

5.2.1 Locally Scoped (Decentralised) Rule Sets

If reconfiguration rules are scoped to subsystems (have a local scope), it is possible that diverse rule sets could exist within a system. Locally scoped rules allow for subsystems to be developed independently using independent rule sets without the need for a high level of discussion / integration with other subsystem engineers / developers.

With many diverse rule sets in existence throughout a system it is entirely possible for processes to become restricted to certain subsystems as diverse rule sets could effectively create subsystem boundaries that processes cannot cross. Research investigating transactions over boundaries given independently-formulated confidential rules is presented in [61]. This research introduces a coalition policy by which all independent subsystem (or enterprise) rules must comply, thus ensuring a certain minimum level of interaction. A similar approach could be taken to subsystem rules for reconfiguration control, whereby a minimum level of reconfiguration is ensured across all subsections, by enforcing all independently formulated rules to abide by a coalition policy. It should be noted that in an approach such as this, if the coalition policy were to require a change then all subsections must update their independently formulated rule sets to meet the requirements of the new coalition policy.

5.2.2 Globally Scoped (Centralised) Rule Sets

If rules are globally scoped, then no fixed subsystem boundaries exist and thus no governing coalition policy is required. Globally scoped rules simplify reconfiguration rule checking as only a single set of rules exist.

Globally scoped rules do not allow for rule sets to be developed in small subsystems and thus will require input from many developers across the entire system / application to establish the rule set required. However, it is expected that a standard core set of reconfiguration rules will exist to control general configuration thrashing which may be tweaked and appropriately augmented for specific applications. This is further discussed in section 5.4.

A change in requirements requiring alterations to be made to a global rule would be simple to implement, though difficult to analyse. The effects on the system will be wide spread and as such may become difficult to certify without significant further testing.

5.2.3 Local (Decentralised) Rule Checking

In a system where reconfiguration rule checking is decentralised, local processing must be conducted to authorise a reconfiguration action. Local rule enforcement would require increased local processing resources, but would not incur the delay introduced when centralised (global) rule checking is used due to network latency, or high load upon the central body.

It is possible, depending upon the rule scope, that decentralised rule checking would require a level of system knowledge to be maintained in order to make decisions upon reconfiguration requests. The work involved in keeping multiple controllers system states in sync could be prohibitively expensive in terms of processing when taking into account the number of system state changes and the quantity of controllers requiring synchronisation.

An alternative to keeping decentralised rule checking controllers in sync is to allow controllers to query individual processes to establish the system knowledge required in order to make decisions. The time taken to query processes could make this approach impractical, though it would depend upon the real-time requirements of the system. The query action would most likely have to occur as part of a transaction; as if a controller has to make many queries to many processes in order to make a decision then it is possible for a process already queried to have changed state before a decision has been made. Restrictions would have to be placed upon the size of a given transaction to ensure that the system does not come to a halt when a reconfiguration request is made. This is similar to the approach taken by Kramer and Magee [65] where all affected nodes and their neighbours are required to be in a quiescent state before changes can be made. While a node is in a quiescent state, it is required not to initiate communication. This ensures that nodes directly affected by a change will not receive requests during the course of a change.

Another alternative to keeping decentralised rule checking controllers in sync is to allow controllers to maintain the state for a given subsystem and query other controllers to get other subsystems data as required. This method would still require queries to be part of a transaction to stop queried data changing before a decision is made, though less queries should be required than would be required if individual processes were to be queried and restrictions could be put in place to ensure that entire systems are not brought to a halt when reconfiguration requests are issued.

Within a decentralised rule checking environment each controller is a single point of failure; as a single controller exists within each subsystem. In the event of a controller failure it is likely that the controllers' subsystem would have to remain static from that point forward. If a controller were to become "faulty", and not fail, the knock on effects would then filter through the system, both in terms of greater load if requests for system state are required and also in terms of reconfiguring processes which invalidate rules within other subsystems.

To stop "faulty" controllers adversely affecting the system a fault tolerant technique must be used. It has been said that "[t]he starting point for all

fault tolerant strategies is the detection of an erroneous state... [t]hus the success of any fault tolerant system will be critically dependant upon the effectiveness of the techniques for error detection" in [32]. Replication error checks could be used to ensure faulty controllers cannot adversely affect the system, thus each controller's decisions should be checked by at least one other controller to ensure that the same decision would be made by the other controller. This will increase the load on all controllers. Other approaches to removing the single point of failure could include having a dual redundant or triplex voting reconfiguration controller mechanism within each subsystem, or even having redundant subsystems. The decision as to the technique used to remove the single point of failure will depend on resources available and the level of criticality of the subsystem and application as a whole.

The approaches outlined for the removal of the single point of failure will not eradicate design failures or imperfect software. If design failures are expected in a system, then replication should be done using diverse software implementations. Diverse software designs have been shown to suffer from co-dependence, which basically means that common faults can appear in diverse implementations often due to specifications. Further information on studies which have been conducted on co-dependence of diverse implementations can be found in [66].

5.2.4 Global (Centralised) Rule Checking

If reconfiguration rule checking is conducted centrally, then reconfiguration can take place only if a central body authorises it. Centralised rule checking would require the centralised controller to have reasonable knowledge of the system state in order for it to make informed decisions. It seems impractical for a centralised controller to query multiple processes at run-time to establish the system knowledge required to make decisions since the high volume of requests occurring would introduce a significant delay.

The system state held within the central controller would most likely be maintained in a simple internal structure. The level of knowledge a centralised controller must possess in order to make decisions is dependent upon the rule definitions. For example, if rules allow developers to state that two processes should not be collocated, then the centralised controller must know where processes are located in order to decide if a move can take place and still preserve the rule.

Since global rule checking requires controllers to have knowledge of the system structure, it will be important to ensure that the controllers can never end up with an invalid system state model. If a controller ends up with an

invalid system state then decisions may be made which break the rules of the system. It should be relatively simple to ensure a valid system state is maintained provided the controller starts with a correct initial system state, and assuming the controller only executes the permitted reconfiguration operators / primitives. If the controller is the only body which can allow a configuration change then it can ensure it updates its state upon each reconfiguration occurrence. However, a centralised controller would also need to know about failures, since a remote process could fail to reconfigure even though it has been given permission to do so. If a centralised controller updates its state on the assumption that the reconfiguration has actually occurred once it has issued permission then the state could become invalid.

In a centralised approach, the requests for reconfiguration arise locally and must be authorised centrally. The events that trigger reconfiguration will often be failures - and their occurrence is naturally locally-detected, which has to then be propagated to the central controller. Problems could arise from a subsystem not notifying the controller of a failure and although this will not affect configuration thrashing it may well cause deadlines to be missed and thus system designers must ensure that failures are detected.

A centralised controller can be used to enforce locally scoped rules or globally scoped rules. To apply locally scoped rules using a central controller, the controller must know all of the subsystem rule sets which exist within the system and also which segments of the system each rule set applies to.

A centralised approach to reconfiguration rule checking inherently introduces a single point of failure in the controller. As discussed in section 5.2.3, a "faulty" controller could allow system states which are not valid, and if the central controller were to fail altogether then it is likely that the system would have to remain static from that point forward. To eradicate this single point of failure it is recommended that replication error checking be used; multiple controllers must exist to check that the decisions made by the controller are correct and to take over should the primary fail. This method will eradicate the single point of failure and also protect the system from faulty controllers, though as discussed in section 5.2.3 this solution will not eradicate design failures or imperfect software.

5.2.5 Further Rule Set Discussion

Rule sets could be dynamic (able to evolve over time), providing advantages, such as simplified upgrades. Newly installed components could add additional rules to the system, thus ensuring the properties they require

are enforced when reconfiguration occurs. Evolving rule sets would be difficult to certify, as it is entirely possible that a corrupt component could add a rule which could stop the system from functioning correctly, or a component could add a rule which contradicts an already existing rule. For instance if the following rules existed in a system, *A must loc(B)* and *A must loc(C),* and then a third rule was added as follows *B must* ¬ *loc (C)*, then the system would have a contradictory (inconsistent) rule set.

Consistency is a well-formedness condition for constraint sets. An inconsistent set of constraints does not admit any valid system state. It is important that a set of system rules can be checked for inconsistencies even when not evolving; if the rule expressions are complex, it is possible for two rules to seem to support each other, but in fact contradict each other, though this should be detected during testing. A restricted notation in which to express rules could help to lower rule complexity, thus reducing the chances of accidental rule conflict creation. Many inconsistencies should appear during testing, they would manifest themselves as deadlocks in many cases.

Dynamic rule sets could themselves become susceptible to configuration thrashing, thus the use of dynamic rule sets will not be developed further within this thesis, even though they could potentially provide many novel benefits. If a temporally predictable run-time consistency checking algorithm can be developed for rule sets then further research in this area would almost certainly be worth progressing.

Reconfiguration could be co-operative or forced. A co-operative approach would request processes to reconfigure. In a forced approach the system would interrupt the appropriate process regardless of its current task / state. With the forced approach, the process being reconfigured could be milliseconds from finishing a major cycle and thus essentially waste the progress it has made up to the point of reconfiguration, as it has not had time to output its results or store them in persistent memory. However, if the process is dealt with in a co-operative manner, then it is possible for the process to not respond to the required reconfiguration request and thus prevent the necessary reconfiguration from taking place. If it could be guaranteed that a process would reconfigure within a certain deadline from a reconfiguration request being made, then a co-operative approach would be preferable; however, this is impossible to guarantee as a process could freeze or have a backlog of buffered inputs to process before processing the reconfiguration request.

A hybrid approach could be taken whereby a co-operative request is made and if no action is taken by the process before a given deadline, then a forced approach is taken. This approach would allow for a higher level of process progress whilst also ensuring reconfiguration eventually occurs,

though the worst case execution time for this is much greater than simply using a forced approach. Though this approach is interesting, it has not been explored further within the thesis as the worst case execution time is increased over and above the forced approach and it also has no method of ensuring that a major cycle wouldn't have just taken a few more milliseconds to complete before reconfiguration was forced.

5.2.6 Rule Expression / Predictability

For a rule based reconfiguration control system to be suitable for real-time mission-critical systems, it is necessary for the decision making algorithm to be predictable, both in terms of the outcome and the time taken to produce the outcome. In order to create a temporally predictable decision making algorithm it is necessary for the rules themselves to have a limited notation, and to ensure that the complexity of the rules is restricted.

Existing rule based control research, such as Law Governed Interaction (LGI) [58-63], does not address the issue of temporal predictability. In [58] it is stated that "enforcement must be reasonably efficient", however "reasonably efficient" is not enough to ensure deadlines are met. Also since any number of LGI rules can be placed within the law of the system, and the rules can be of varying complexity (LGI rules are expressed in an unrestricted programming language), it is almost impossible to carry out worst case timing analysis.

Restrictions in notation could simply outline a set of operators by which rules can be specified. These operators could include the candidate set of reconfiguration operators specified in chapter 2. These operators would require further extensions to include operators such as *collocated(proc1, proc2)*, which would be a logical operator to check if two processes are located on the same processor(s).

Further research is required to examine the best method of restricting rule set complexity. Research into rule set complexity could also develop accurate worst case execution time calculations. To calculate worst case execution time many difficult issues would require further examination, such as can rules be applied iteratively?, how long do given operators take to execute?, and does the order of the rules matter? – does it change the outcome or the time taken to get to the outcome?

5.2.7 Core Malfunction and Reconfiguration Thrashing Restriction Rules

Within most (if not all) reconfigurable real-time applications a standard core set of rules may exist to stop malfunctioning processes from adversely affecting the system, as well as generally controlling configuration thrashing. There are many different rules that could be imposed on a system regardless of whether rule-checking is conducted centrally or locally. Some examples are: a process cannot reconfigure more than once in a given time period, or processes may only reconfigure if they do not communicate with another process (or processes).

To develop rules for configuration thrashing control it is important to consider the definition of configuration thrashing. In Chapter 3 configuration thrashing is defined as "...occurring when one or more configuration overlaps occur. The number of configuration overlaps that can be tolerated in a given time period or in a given sequence is application dependent and possibly even mode dependent...". A configuration overlap occurs when two subsequent reconfiguration requests are acted upon without a "sufficient interval" between them.

Given this definition for configuration thrashing it is logical for one of the core rules to state that no more than x configuration overlaps may occur in a given time period, where x is the number of overlaps that can be tolerated in the given application. For this to be enforced a configuration overlap would need to be defined for the specific application – i.e. the sufficient interval would require a definition. This rule could exist in all systems but would require variables to be specified. The operator for this could be specified as follows:

ConfOver(x, y, z)

Where x is the maximum number of configuration overlaps which may occur, y is the interval in which the x configuration overlaps may occur, and z is the "sufficient interval" between reconfigurations to not be classified as a configuration overlap. In many applications this operator may require an extension in order to allow an additional variable to specify the process, thus allowing multiple processes to have different configuration thrashing definitions. Pseudo code showing how the *ConfOver* operator could be applied is shown below:

```
If time since last successful reconfiguration request > z then
       Reconfiguration is not a configuration overlap and is permitted and
recorded
Else
       //reconfiguration is an overlap
       If x overlaps have occurred in y interval then
              Reconfiguration denied
       Else
              Reconfiguration overlap recorded
              Reconfiguration permitted
```

Preconditions for the *ConfOver* operator may be necessary, as all of the variables (x, y, and z) must be greater than 0. Also it is expected that $(x+1)*z$ should be less than or equal to y, as if this is not the case then the *ConfOver* operator cannot guarantee that configuration thrashing will not occur. The guarantee cannot be made as in theory the system could make no progress for just under $x*z$ units of time in the worst case and the sufficient interval (z) would be required again to make the required level of progress. However, $(x+1)*z <= y$ should not be made a precondition, as it is possible in some systems for it to be guaranteed that if a reconfiguration is to be made, then it will occur very soon after the last reconfiguration or not at all. In a system such as this it is possible for $(x+1)*z$ not to be less than or equal to y and still ensure configuration thrashing will not occur.

As the pseudo code for the *ConfOver* operator shows, this rule would require the controller to keep a history of configuration changes / overlaps in order to apply the rule, which could prove resource intensive. Within the CSP models presented in Chapter 3 a history was maintained by which to check for configuration overlaps and thus configuration thrashing. In the approach taken within those models, events were removed from the history when they became stale (i.e. were no longer of use when analysing for configuration thrashing). A similar approach could be taken within the controller to condense the resource usage records; however, this would still take up computational resources and as such a method must be available to analyse resource requirements. This method may also affect temporal predictability.

Creating generic rule(s) to stop configuration thrashing for all systems without variables being added would effectively require all systems to be static. In some systems it may be that reconfiguration can occur once an hour, yet in others it may be acceptable to reconfigure every ten seconds.

Another approach using rule sets to control reconfiguration thrashing would be to alter the rule set at runtime. Configuration thrashing could then be avoided by for example, adding rules to state that the reconfigured process must remain static for a period of time after a successful reconfiguration or a number of consecutive successful reconfigurations. However, this could alter the time taken to analyse the rules in existence. Also the rule should be

removed once it has exceeded the time that the process must remain static (become stale) which would incur a processing overhead. As discussed in section 5.2.5, dynamic rule sets could themselves become susceptible to configuration thrashing, thus this approach is not adequate.

It can be argued that to stop a process from reconfiguring after one or more successful reconfigurations is wrong, as if a process moves to avoid a failure, and then encounters another failure, why is that it's fault and why shouldn't it be allowed to move again? This is a valid argument and all approaches to restricting configuration thrashing can be said to fall foul of this argument. However, if many successive failures have occurred / are occurring then either the software itself has an issue which cannot be rectified through reconfiguration or the underlying hardware has major issues, thus restricting reconfiguration will most likely not worsen the situation, but will solve the problem of configuration thrashing .

To stop malfunctioning processes from adversely affecting the system a set of operators should be developed to constrain the system. An example of this would be an operator to keep two processes collocated. If an operator such as this were used, then a subsequent move operator, of any type, would cause both processes to move or deny the reconfiguration request to ensure they both remain collocated. The choice as to whether the processes are moved or the request is denied is dependent upon the logic residing in the reconfiguration controller.

Other examples of possible constraint operators could include:

- *Static(proc)* – ensures the specified process cannot move.
- *NumHops(proc1, proc2, x)* – ensures that communication between two processes is kept to within x hops. An operator such as this would keep communication latency low.
- *LocateOnlyOn(proc,[loc1,loc2,...])* – ensures that the specified process only ever executes in the locations specified. This maybe an important operator if some processors don't have instruction sets that a process requires.

Operators to stop certain types of reconfiguration may also be beneficial, as some processes may be designed to be mobile, but not duplicated, or other processes may be designed to be synchronised with, but not be mobile.

The core set of rules for all systems to stop malfunctioning processes from adversely affecting the system, as well as generally controlling configuration thrashing would include at least one rule utilising the *ConfOver* operator and any restraints designed into processes should be specified in rules utilising the constraint operators, i.e. each process that

should remain static should have a rule in place to ensure that it remains static and processes that should be collocated should have rules in place to ensure they remain collocated.

5.2.8 Rule Set Reconfiguration Control Demonstrator

A demonstrator has been developed to show that the solution outlined using rule sets can restrict reconfiguration sufficiently to eliminate configuration thrashing. The demonstrator also allows experimentation to be conducted. Experimentation can be conducted not only for rule sets, but also for scenarios where developers control reconfiguration themselves; the controller can be disabled and reconfiguration operators can be accessed directly. Section 5.3 discusses the use of the demonstrator and the experimentation conducted with controllers disabled.

The demonstrator has been built using Java RMI. Java RMI provides a means of invoking methods on remote Java objects. Methods can be invoked from separate Java virtual machines, possibly on different hosts. Although Java is not suitable for real-time applications, the demonstrator uses Java as the demonstrator itself was not intended to look into the problem to the level at which a real-time language would be required. The demonstrator was only intended to highlight that the proposed solution can eliminate configuration thrashing and to allow experimentation to be conducted.

Java is not suitable for use in real-time applications as it does not respond reliably and predictably to real-world events. One of the reasons for this is that Java contains a garbage collection system which takes care of freeing dynamically allocated memory that is no longer referenced. Programmers in a garbage-collected environment have less control over the scheduling of CPU time devoted to freeing objects that are no longer needed. If a real-time event occurred during or just before garbage collection, then timing would be unpredictable. Java have outlined a Real-Time Specification for Java (RTSJ) [67] which enables developers of real-time applications to take full advantage of the Java language while maintaining the predictability of current real-time development platforms.

Within the demonstrator both rule scope and rule validation occur centrally. The decision as to whether rule scope and rule checking / validation should occur in a centralised or decentralised manner is an application specific decision and thus an arbitrary decision was made for the demonstrator. Section 5.2 discusses these options in further detail.

To allow reconfigurable processes to be controlled by a central controller all reconfigurable processes must extend a *ReconfigProcess* interface which defines several functions that each process must implement including *delProcess()*, *getData()* and *setData()*. Using these functions the central controller can reconfigure processes, query and set processes internal states, as well as delete processes. No operators are developed within the demonstrator to allow for the synchronisation of process stacks, though this could have been facilitated by pausing the executing thread and passing it to a new process to continue. As discussed is Chapter 2 the stack represents the instruction stack (including the current position within the instruction stack).

The demonstrator contains implementations of the following small set of candidate operators:

- *MoveProcDelFirst*
- *MoveProcDelAfter*
- *MoveProcWState*

The demonstrator is coded to execute on a single machine, though tests have been conducted between multiple machines and there is no functional difference, though there is slight performance degradation. All services are bound to the *rmiregistry*, which allows users to locate services needed. The *rmiregistry* provides a means of location transparency.

Each reconfigurable process must provide a method of generating instances of themselves. Within Java this is achieved using factories which essentially generate instances of processes upon demand. A *ProcessFactoy* interface has been defined and each reconfigurable process must provide an implementation which extends this interface. Process factories must be bound to the *rmiregistry* to allow processes to be generated on remote hosts.

Factories for creating reconfigurable processes could exist in a single location or could be duplicated in many locations. There are advantages and disadvantages to duplicating factories to many locations. One advantage of duplicating factories is that multiple factories could spread the load on the system for process creation. Another advantage is that if a processor were to be damaged with a process factory upon it, then the other process factories could continue to allow process creation with no need to recreate the failed factory. If factories are not duplicated and a processor were to be damaged that contained a process factory then no more process generation can occur for that process type until a new instance of that factory can be brought online (if one can be brought online). A disadvantage of factory duplication is that a higher level of processing resource is required to support multiple factories

An example process has been developed within the demonstrator which is susceptible to configuration thrashing. The example process named *ReconfigProcessImpl.java* attempts to complete a basic cycle which is defined in *MainThread.java* and is shown in definition 5.2.8.1. The cycle can be interrupted through reconfiguration.

Definition 5.2.8.1

```
public void run()
{
  int i = 0;
  for (i=0; i<10; i++)
  {
    try
    {
      Thread.sleep(5000);
    } catch (Exception e)
    {}
    System.out.println(i);
  }
  System.out.println("cycle complete");
}
```

Given definition 5.2.8.1 *ConfOver* rules were be placed on the system to restrict reconfiguration. The example process was assigned many different deadlines to complete and from these many different *ConfOver* rules were produced. In each case it was found that with some basic calculations a set of values could be produced for *ConfOver*. In each case it was found that if $(x+1)*z$ was ensured to be less than or equal to y, then configuration thrashing was stopped. Though further experimentation showed that in cases where it was guaranteed that if a reconfiguration was to be made, then it would occur very soon after the last reconfiguration or not at all then *ConfOver* rules could be used which did not ensure that $(x+1)*z$ was equal or less than y and still stop configuration thrashing. This further demonstrates that although (x+1)*z should be less than or equal to y in order to guarantee the elimination of configuration thrashing, it should only be used as a guideline and not a precondition.

Experimentation was also conducted to see if all of the variants of the candidate set of operators were required. The demonstrator includes implementations of many of the variants of the *MoveProc* operator (*MoveProcWStateAndSync* is excluded) and it was found that the *MoveProcDelFirst* and the *MoveProcDelAfter* were in the main the same, though there are scenarios in which the two would both be required. It was also found that a process could be queried for its internal state, then be reconfigured, and then have its internal state reinstated, thus not requiring the *MoveProcWState* operator, but this is a lot more confusing for

developers and also introduces the chance of stale states being inserted into reconfigured processes.

The demonstrator clearly highlighted through the use of the example process how severe the effect configuration thrashing can have upon real-time applications. With no rules placed in the system and reconfiguration operators triggered over and over again, the example process could make no progress at all; in fact in sever cases it didn't even manage to make a single output in more than 10 minutes. The demonstrator showed that configuration thrashing can be eliminated using a central controller even when reconfiguration operators are triggered over and over again, thus even faulty processes can be constrained. However, further research and development is required to deal with issues such as how to cope with high load and how to ensure that when large numbers of rules exist, the algorithm used within the controller is temporally predictable. The demonstrator in its present state has only been tested with a small number of rules. Appendix D contains the complete source code for the demonstrator.

5.3 Mechanisms Allowing Developer to Control Reconfiguration

Many types of mechanism could be provided to developers in many forms, for example mechanisms could be provided in the form of a class for inclusion in source code, or could be provided as a service (possibly a middleware service). The form in which the mechanisms are provided should make little difference to functionality, but could make a difference to the time taken to execute a mechanism. The form in which a mechanism is provided will affect the degree of separation between the mechanisms and the reconfigurable processes. If the reconfiguration mechanisms are compiled into the source code, then the process is tightly coupled to the mechanisms, but if for a reconfiguration mechanism to be provided as a service then they would be more loosely coupled.

Mechanisms could be provided to developers allowing them to declare windows of opportunity within source code for processes to reconfigure. This could be further developed to allow the specification of reconfiguration operator(s) which may be used during the period the window of opportunity is open. For example a developer could open a window of opportunity that enables process replication, but not process migration. Chapter 2 introduced a candidate set of reconfiguration operators.

As described, the demonstrator discussed in section 5.2.8 was developed in such a way that the controller could be disabled and reconfiguration operators could be accessed directly, thus allowing experimentation to be

conducted in a scenario where developers control reconfiguration. The experimentation conducted clearly highlighted that if processes are to be able to initialise reconfiguration for other processes, then some form of additional acceptance is required. If the reconfiguring process cannot accept or decline the reconfiguration action, then it will be very difficult for developers to ensure that their processes do not become susceptible to configuration thrashing, as processes could trigger configuration thrashing in one another. The method described above could be a solution to this; whereby processes declare windows of opportunity for others to reconfigure them and if no window is presently available then reconfiguration cannot occur.

Experimentation with the demonstrator also highlighted the benefit of making the reconfiguration operators available via a service (or third party process), as during the experimentation the reconfiguration operators required a small amount of tweaking and this meant that the service alone had to be recompiled and not the individual processes being reconfigured or triggering reconfiguration. If the operators were compiled into the processes, then much more recompilation would have been required. Advantages in terms or runtime upgrades can also be achieved as during the recompilation only the reconfiguration service was are offline and other processing can be continued.

Giving developers control over reconfiguration on a process by process basis is a novel approach, provided that guidance can be provided. Without guidance developers may not know when reconfiguration should take place or more importantly is "safe" to take place. Developers may be tempted to develop more static systems than necessary, thus not gaining the full benefits that reconfiguration can provide. Also allowing developers to control reconfiguration introduces the likelihood of human error.

5.3.1 Difficulties Providing Guidance for Developers

Providing general guidance to developers on the matter of when reconfiguration can safely occur without allowing configuration thrashing to happen is a difficult task as different types of systems will require different logic to be applied. However, a starting point can be provided by ensuring that system developers fully understand the definition of configuration thrashing. Armed with the definition developers should be better equipped to avoid configuration thrashing. However, this is not sufficient as it can be very difficult to decide whilst designing / developing software whether a configuration overlap could have occurred before a particular point in the software, or if a "sufficient interval" will have definitely occurred at a given point in the software.

As different logic is required for different process types, this section explores the method used to define the *ConfOver* rules for the example process (as described above). From the method used to define these rules, we attempt to use the same logic to develop a method of restricting the processes in source code.

In the example process defined in the demonstrator, it was relatively simple to define the variables which must be plugged into the *ConfOver* rule in order to allow a controller to stop configuration thrashing. This was done by reviewing the cycle length and calculating variables using the $(x+1)*z$ must be less than or equal to y equation. However, it is not simple to use the same logic in source code, unless a history of reconfiguration actions taken is made and windows of opportunity are opened based upon the assessment of the actions taken.

To allow each process to track its previous reconfigurations could mean that an element of the processes state would be transferred even if the process migration operator used to conduct the reconfiguration did not require / request for the state to be transferred, as the history of the reconfiguration actions will most likely be held in its internal state. Transferring reconfiguration history between processes upon reconfiguration could create issues with temporal predictability and also contradict some of the operator's definitions. An alternative method to tracking reconfiguration history is to have a process / service available to track reconfigurations and maintain histories for processes to use when making decisions, however this method has a few downfalls: firstly the service / process could fail causing all reconfigurable processes to have to take a default action, which will most likely be to remain static, and secondly this approach could add significant network lag depending upon the number of queries made and level of history to be maintained.

To decide when reconfiguration can occur within the source code for the example process in the demonstrator without the maintenance of some form of history of reconfiguration events, it would only be safe to allow reconfiguration to occur every other loop through the code. Note that though the example process in the demonstrator only completes one cycle, it is designed to represent a cyclic process whereby it simply restarts the cycle once complete. This will mean that the process must maintain a reconfigured state between cycles and thus even though a full history of reconfiguration events is not required, some information relating to recent reconfiguration is required. This approach is quite restrictive and given a process with a definition of configuration thrashing whereby reconfiguration can occur 12 times before it must be restricted would require the process to essentially over restrict reconfiguration or start to maintain a number of

reconfigurations that have occurred which is essentially starting to maintain a history of reconfiguration events.

The example process reviewed is relatively simple and yet provides difficulties when attempting to provide guidance, thus more complex processes whereby branching occurs and synchronisation can occur upon shared variables would make the provision of advise almost impossible and developers are almost certain to over restrict reconfiguration in order to relieve the level of complexity they are faced with.

5.4 Summary

This chapter has explored potential run-time solutions to configuration thrashing. These solutions allow developers to include additional logic / processes in their systems in order to eliminate configuration thrashing. Several options were explored from providing mechanisms for developers to choose when reconfiguration can / cannot occur to a rule based solution.

The explorations described and discussed within this chapter have shown that run-time solutions to configuration thrashing using a rule based approach can work, but at present many problems and imponderables exist for performance analysis. The problems and imponderables for the rule based approach include:

- Rule Set Consistency Checking - An inconsistent set of constraints does not admit any valid system state, thus it is important that a set of system rules can be checked for inconsistencies. A timely method of checking rule set consistency is therefore required.

- Worst Case Execution Time Analysis – It is important in a real-time application that the worst case execution time can be analysed to ensure that deadlines are met. To calculate worst case execution time for the rule checking algorithm many difficult issues would require further examination, such as can rules be applied iteratively?, how long do given operators take to execute?, and does the order of the rules matter? – does it change the outcome or the time taken to get to the outcome?

Allowing developers to control reconfiguration within source code gives developers a high level of control, however without guidance developers may not know when reconfiguration should occur and thus are likely to over restricted reconfiguration. Also allowing developers to control reconfiguration introduces the likelihood of human error. Providing guidance to developers has proven difficult, as different types of systems

require different logic to be applied. Guidance can be provided for specific examples, but to categorise all process possibilities and provide guidance could be a thesis in its own right as there are potentially infinite numbers of system possibilities in which guidance may be required. This thesis has therefore only provided advice for the example process in the demonstrator.

Chapter 6

Case Study

To review the effectiveness of both the models outlined for configuration thrashing, and also the software solution outlined, a small case study has been developed. This case study is focussed upon battlefield surveillance using multi sensor data fusion [68, 69]. The ability to rapidly detect and identify potential targets both fixed and mobile from multiple sensor inputs is a critical function in modern warfare. Sensor fusion has been used in major military weapons systems, such as the U.S. Navy's Cooperative Engagement Capability (CEC), a Raytheon built system that enables ships and aircraft to combine radar data for improved defences against attack from aircraft and cruise missiles.

A fully functional sensor network can perform many military-related intelligence missions in synchronization and harmony with human agents in the battlefield. A typical sensor network may consist of many different types of sensors such as satellites, radars, ground sensors (magnetic, acoustic, and seismic), and infra red imaging. An aim for research into sensor fusion is to produce cheap reliable sensors that are disposable.

The key to any battlefield surveillance system is the rapid generation of target information. Targets need to be assessed as quickly as possible in order to guide troops accurately and ensure that missiles are aimed only at enemy targets. If data is not processed quickly enough then troops could be put in dangerous situations and / or missiles could be launched against friendly or non-hostile targets. Based on this it is possible for a reconfigurable multi-sensor data fusion system to suffer from configuration thrashing and as such forms the basis for this case study.

Within this case study multiple sources of sensor data are required to get data to the processing unit within a periodic timeframe and the processing unit itself is required to process the data within a finite time frame otherwise the data used for decision making will be outdated and dangerous scenarios can no longer be guaranteed to be avoided. The case study comprises of three software components which are: a radar sensor, a ground sensor, and the main sensor fusion and battlefield decision making component.

The rest of this chapter is structured as follows. First section 6.1 outlines the component functionality for use within the case study. Section 6.2 explores the formal approach taken to the case study. Section 6.3 introduces and examines the software approach taken to the case study. Lastly, section 6.4 presents a case study discussion.

6.1 Component Design

In order to model or develop the software components for this case study, it is important to understand the functionality of those components. Here we outline each components functionality and describe which parts have been abstracted away for the purposes of this case study. This case study does not require perfect models or implementations as we are only attempting to capture the main elements and also no appropriate hardware is available for the sensor data to come from, thus this will have to be simulated.

Within the case study it is assumed that data communications will always be reliable and instant or so close to it that it is negligible. Though "lossy" channels [70] could be added to the CSP to model a certain level of data loss and similar techniques could be used in the software solution.

The usage of fixed processing times has been used within this case study as it would be assumed that these would be the worst case execution times and as such can be used to ensure that the system cannot thrash. However, variant times between fixed bounds could easily be added to the models outlined in this chapter and also the software developed.

6.1.1 Radar Sensor

A radar system has a transmitter that emits either microwaves or radio waves that are reflected by the target and detected by a receiver, typically in the same location as the transmitter. Although the signal returned is usually very weak, it can be amplified enabling the radar to detect objects at ranges where other emissions, such as sound or visible light, would be too weak to detect.

Thus the radar sensor must emit a signal, receive a signal back, process the data received, and then it can send the results to the main decision making component for sensor fusion and decision making algorithms to be run. Depending on the type of signal emitted and the strength of the data received back it may take a while for the data received back to be processed.

It is possible for radar sensor software to reconfigure for many reasons, the most obvious being fault tolerance. Other reasons could include changing the signal processing engine or to adapt how the sensor hardware is being utilised. Moving the component to different hardware could also be for resource reasons.

For the purpose of this case study the sensor data will be simulated and the processing time will be assumed to be static all times. No implementation will be provided for the signal processing as this is irrelevant for this case study. Within the case study it is assumed that an output must be provided to the data fusion and decision making process every 5 minutes. The time taken to receive data after the emition of either microwaves or radio waves is assumed to always be 1 minute and the time taken to process the data is assumed to be a static 1.5 minutes.

6.1.2 Ground Sensor

Ground sensors consist of a variety of sensor technologies that are packaged for deployment and perform the mission of remote target detection, location and / or recognition. Ideally, the ground sensors should be small, low cost and robust, and are expected to last in the field for extended periods of time.

Ground sensors can be designed to locally process target information, such as detection, bearing estimation, tracking, classification and / or identification. They can also be used for reporting battle damage assessment (BDA) in standoff strike scenarios. Ground sensors may consist of a battery-powered, single or multiple co-located sensors, with signal processing capability to analyse target characteristics, and transmit target recognition information to a remote monitoring location.

For this case study we will focus upon an acoustic ground sensor which is similar to the radar sensor in terms of its processing, apart from does not need to emit a signal before receiving a signal. Basically an acoustic ground sensor continually polls for sound signals, once received it processes the data, and then sends the results to the main decision making component for sensor fusion and decision making processing. We also assume that the particular sensor also sends a periodic signal to the main decision making component if no sound is detected.

Ground sensor software will reconfigure very infrequently as in many cases only a single processor will be included in the small device which generally will have no way of utilising other processing hardware. However for the purpose of this case study it is assumed that the sensor is a little more sophisticated and has multiple processors mainly for fault tolerance purposes.

As with the radar sensor, sensor data will be simulated and the signal / input processing time will be fixed. No implementation will be provided for the signal processing as this is not needed for this case study. This sensor will

only reconfigure once data has been sent to the fusion and decision making component and only if it suspects there is an issue.

Within the case study it is assumed that an output must be provided to the data fusion and decision making process every 5 minutes. The time taken to process data is static at 1 minute.

6.1.1 Sensor Fusion & Battlefield Decision Making Component

The sensor fusion and battlefield decision making component is of critical importance as this component must make decisions based on information provided as to where troops are sent and also where weapons will be targeting. Thus it is of great importance that the data this component makes its decisions with are up to date and accurate. It is likely that in a true system the sensor fusion and decision making functionality would be separated into different components allowing for separation of concerns and increased flexability, however for the purposes of this case study they will be considered as one.

It is possible for this component to reconfigure for fault tolerance, improved performance, and also ensuring data continuity. This type of component could also be subject to online upgrades and could possibly reconfigure to cope with changing requirements and logic for battlefield processing.

For the purpose of this case study this component will receive data from the two sensors and make periodic outputs as to the strategy that should be undertaken. The processing time for the data fusion and decision making algorithm will be assumed to be the same at all times. No implementation will be provided for the data fusion or the decision making algorithm as this is not required for this case study.

Within the case study it is assumed that an output must be provided every 10 minutes. The time taken to fuse data and process data are both static at 0.5 minutes and 1 minute respectively.

6.2 Formal Approach

In Chapter 3 two unique CSP models have been specified which enable developers to model their systems / processes and detect configuration thrashing potential. One of the models contains an element of time and the other does not. These models assist developers to engineer configuration thrashing out of their systems.

This section introduces the CSP models representing the three software components for this case study and discusses the issues that had to be overcome both in terms of developing the models, and also in terms of using the models to engineer configuration thrashing out of the system. The timed CSP model for configuration thrashing introduced in section 3.3.2 was used for this case study.

In chapter 3 the models introduced only contained a restricted set of actions. All actions that related to internal process actions were modelled using the event *doa*. In order to model the two sensors the set of actions usable was extended to include the following: *send_to_fusion*, *send_signal*, *recieve_signal*, *start_process_data*, and *end_process_data*. These new event types allow for the model not to contain a high level of information regarding development detail but still capture the main events so that reconfiguration points and timing can be reviewed. As with the model presented in chapter 3 the new actions are hidden at a later stage as they are not needed to check for configuration thrashing.

The radar sensor has been modelled in two parts. First the main *RADAR* definition is used to start the process and allow it to reconfigure or become *RADARWORKLOOP*. *RADAR* and *RADARWORKLOOP* are shown in definition 6.2.1 and definition 6.2.2 respectively. The reason the logic of this process has been broken in two is to simplify the use of the *startup* action after a reconfiguration has taken place.

Definition 6.2.1

```
RADAR = startup ->
                (RADARWORKLOOP
                [] move -> RADAR)
```

Definition 6.2.2

```
RADARWORKLOOP = start_min_wk -> send_signal -> tock -> tock ->
recieve_signal ->
        start_process_data -> tock -> tock -> end_process_data ->
send_to_fusion -> end_min_wk
        -> RADARWORKLOOP
                [] start_min_wk -> send_signal -> move -> RADAR
                [] start_min_wk -> send_signal -> tock -> move -> RADAR
                [] start_min_wk -> send_signal -> tock -> tock -> move ->
        RADAR
                [] start_min_wk -> send_signal -> tock -> tock ->
        recieve_signal -> move -> RADAR
                [] start_min_wk -> send_signal -> tock -> tock ->
        recieve_signal -> start_process_data -> move -> RADAR
                [] start_min_wk -> send_signal -> tock -> tock ->
        recieve_signal -> start_process_data -> tock -> move -> RADAR
                [] start_min_wk -> send_signal -> tock -> tock ->
        recieve_signal -> start_process_data -> tock -> tock -> move ->
        RADAR
                [] start_min_wk -> send_signal -> tock -> tock ->
        recieve_signal -> start_process_data -> tock -> tock -> tock ->
        end_process_data -> move -> RADAR
                [] start_min_wk -> send_signal -> tock -> tock ->
        recieve_signal -> start_process_data -> tock -> tock -> tock ->
        end_process_data -> send_to_fusion -> end_min_wk -> move -> RADAR
```

The radar sensor can take external stimuli to reconfigure and as such it has been assumed that it can reconfigure at any point. The model reflects this. It should be noted that the radar sensor does not allow reconfiguration to take place between the *start_min_wk* and *send_signal* actions as these are seen as being the same thing. The *start_min_wk* event is only contained within the model to ensure that the configuration thrashing definition knows that work has begun. The same holds for the *send_to_fusion* and *end_min_wk* actions.

Within the model a *tock* action has been used to model the passage of 30 seconds of time. As can be seen from the model, if no reconfiguration occurs then it will send an output to the data fusion and decision making component every 2.5 minutes. This means it can in theory easily meet its deadlines.

As the radar sensor can reconfigure at any time via external stimuli then the models have shown that if developed to be completely reconfigurable it can suffer from configuration thrashing. The definition of configuration thrashing used within this model is 2 overlaps in 10 time intervals (5 minutes). This was used as the time taken to complete a successful cycle is 2.5 minutes and thus if we have 2 overlaps inside the 5 minute deadline then there is a possibility that the deadline has not been met. If we have a system that can only reconfigure at the start of its processing then it may be that a larger number of overlaps would be acceptable, however in this case reconfiguration can take place at any point.

A restricted version of the radar process has then been modelled using the information gathered from the modelling of the non-restricted radar. Two

extra definitions have been added to the restricted radar to ensure that after a reconfiguration occurs, another does not until the minimum work is completed. These two definitions are named *RADARNORECONF* and *RADARWORKLOOPNORECONF* and are shown in definitions 6.2.3 and 6.2.4.

Definition 6.2.3

```
RADARNORECONF = startup ->
                RADARWORKLOOPNORECONF
```

Definition 6.2.4

```
RADARWORKLOOPNORECONF = start_min_wk -> send_signal -> tock -> tock ->
       recieve_signal -> start_process_data -> tock -> tock ->
       end_process_data -> send_to_fusion -> end_min_wk -> RADARWORKLOOPREST
```

The main radar process and the *RADARWORKLOOP* have both been modified to ensure that when a reconfiguration occurs that they go to the *RADARNORECONF*. The modified radar definitions are shown in definitions 6.2.5 and 6.2.6.

Definition 6.2.5

```
RADARREST = startup ->
              (RADARWORKLOOPREST
              [] move -> RADARNORECONF)
```

Definition 6.2.6

```
RADARWORKLOOPREST = start_min_wk -> send_signal -> tock -> tock ->
        recieve_signal -> start_process_data -> tock -> tock ->
        end_process_data -> send_to_fusion -> end_min_wk -> RADARWORKLOOPREST
                [] start_min_wk -> send_signal -> move -> RADARNORECONF
                [] start_min_wk -> send_signal -> tock -> move ->
        RADARNORECONF
                [] start_min_wk -> send_signal -> tock -> tock -> move ->
        RADARNORECONF
                [] start_min_wk -> send_signal -> tock -> tock ->
        recieve_signal ->move -> RADARNORECONF
                [] start_min_wk -> send_signal -> tock -> tock ->
        recieve_signal -> start_process_data -> move -> RADARNORECONF
                [] start_min_wk -> send_signal -> tock -> tock ->
        recieve_signal -> start_process_data -> tock -> move -> RADARNORECONF
                [] start_min_wk -> send_signal -> tock -> tock ->
        recieve_signal -> start_process_data -> tock -> tock -> move ->
        RADARNORECONF
                [] start_min_wk -> send_signal -> tock -> tock ->
        recieve_signal -> start_process_data -> tock -> tock -> tock ->
        end_process_data -> move -> RADARNORECONF
                [] start_min_wk -> send_signal -> tock -> tock ->
        recieve_signal -> start_process_data -> tock -> tock -> tock ->
        end_process_data -> send_to_fusion -> end_min_wk -> move -> RADARREST
```

The ground sensor is much less reconfigurable than the radar sensor. In fact it has been assumed within the model that reconfiguration will only occur after an output has been made to the data fusion and decision making process. Also this type of sensor is unable to receive external stimuli to reconfigure; it is an internal decision if issues are detected.

As with the radar sensor, the ground sensor has been modelled in two parts to simplify the use of the *startup* action after a reconfiguration has taken place. First the main *GROUND* definition is used to start the process and then it becomes *GROUNDWORKLOOP*. *GROUND* and *GROUNDWORKLOOP* are shown in definition 6.2.7 and definition 6.2.8 respectively.

Definition 6.2.7

```
GROUND = startup ->
                GROUNDWORKLOOP
```

Definition 6.2.8

```
GROUNDWORKLOOP = start_min_wk -> recieve_signal -> start_process_data ->
        tock -> tock -> end_process_data -> send_to_fusion -> end_min_wk ->
        RADARWORKLOOP
                [] start_min_wk -> tock -> recieve_signal ->
        start_process_data -> tock -> tock -> end_process_data ->
        send_to_fusion -> end_min_wk -> RADARWORKLOOP
                [] start_min_wk -> tock -> tock -> recieve_signal ->
        start_process_data -> tock -> tock -> end_process_data ->
        send_to_fusion -> end_min_wk -> RADARWORKLOOP
                [] start_min_wk -> tock -> tock -> tock -> recieve_signal ->
        start_process_data -> tock -> tock -> end_process_data ->
        send_to_fusion -> end_min_wk -> RADARWORKLOOP
                [] start_min_wk -> tock -> tock -> tock -> tock ->
        recieve_signal -> start_process_data -> tock -> tock ->
        end_process_data -> send_to_fusion -> end_min_wk -> RADARWORKLOOP
                [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
        recieve_signal -> start_process_data -> tock -> tock ->
        end_process_data -> send_to_fusion -> end_min_wk -> RADARWORKLOOP
                [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
        tock -> send_to_fusion -> end_min_wk -> RADARWORKLOOP
                [] start_min_wk -> recieve_signal -> start_process_data ->
        tock -> tock -> end_process_data -> send_to_fusion -> end_min_wk ->
        move -> GROUND
                [] start_min_wk -> tock -> recieve_signal ->
        start_process_data -> tock -> tock -> end_process_data ->
        send_to_fusion -> end_min_wk -> move -> GROUND
                [] start_min_wk -> tock -> tock -> recieve_signal ->
        start_process_data -> tock -> tock -> end_process_data ->
        send_to_fusion -> end_min_wk -> move -> GROUND
                [] start_min_wk -> tock -> tock -> tock -> recieve_signal ->
        start_process_data -> tock -> tock -> end_process_data ->
        send_to_fusion -> end_min_wk -> move -> GROUND
                [] start_min_wk -> tock -> tock -> tock -> tock ->
        recieve_signal -> start_process_data -> tock -> tock ->
        end_process_data -> send_to_fusion -> end_min_wk -> move -> GROUND
                [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
        recieve_signal -> start_process_data -> tock -> tock ->
        end_process_data -> send_to_fusion -> end_min_wk -> move -> GROUND
                [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
        tock -> send_to_fusion -> end_min_wk -> move -> GROUND
```

As the ground sensor only reconfigures after the minimum level of processing has completed, it is by definition incapable of configuration thrashing as configuration thrashing is defined as x overlaps in a given time period and an overlap occurs when two subsequent reconfiguration requests are acted upon without a "sufficient interval" between them (i.e. where the end_min_wk has not occurred). The ground sensor can only reconfigure after the minimum level of processing has completed.

The radar sensor and the ground sensor are very different in terms of configuration thrashing properties even though they have a similar cyclic behaviour and effectively provide similar data to the data fusion and decision making component. The difference in reconfiguring as a result of external stimuli creates the significant differences between these sensor types.

In order to define the fusion and decision making process the action set was extended further. The following actions were added: *output_decision*, *start_fuse_data*, and *end_fuse_data*. These actions are only needed to show

the process functionality and are hidden at a later stage as they are not needed to check from configuration thrashing.

The fusion and decision making component has been modelled in three separate pieces to simplify the model in terms of number of separate options available. The three components are *FUSION*, *FUSIONWORKLOOP*, *RECONFIGFUSIONWORKLOOP* and are shown in definitions 6.2.9, 6.2.10, and 6.2.11.

Definition 6.2.9

```
FUSION = startup ->
              FUSIONWORKLOOP
```

Definition 6.2.10

```
FUSIONWORKLOOP = start_min_wk -> RECONFIGFUSIONWORKLOOP
               [] start_min_wk -> tock -> RECONFIGFUSIONWORKLOOP
               [] start_min_wk -> tock -> move -> FUSION
               [] start_min_wk -> tock -> tock -> RECONFIGFUSIONWORKLOOP
               [] start_min_wk -> tock -> tock -> move -> FUSION
               [] start_min_wk -> tock -> tock -> tock ->
       RECONFIGFUSIONWORKLOOP
               [] start_min_wk -> tock -> tock -> tock -> move -> FUSION
               [] start_min_wk -> tock -> tock -> tock -> tock ->
       RECONFIGFUSIONWORKLOOP
               [] start_min_wk -> tock -> tock -> tock -> tock -> move ->
       FUSION
               [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
       RECONFIGFUSIONWORKLOOP
               [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
       move -> FUSION
               [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
       tock -> RECONFIGFUSIONWORKLOOP
               [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
       tock -> move -> FUSION
               [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
       tock -> output_decision -> end_min_wk -> FUSIONWORKLOOP
```

Definition 6.2.11

```
RECONFIGFUSIONWORKLOOP = recieve_signal -> start_process_data -> tock ->
        tock -> end_process_data -> start_fuse_data -> tock -> end_fuse_data
        -> output_decision -> end_min_wk -> FUSIONWORKLOOP
                [] move -> FUSION
                [] recieve_signal -> move -> FUSION
                [] recieve_signal -> start_process_data -> move -> FUSION
                [] recieve_signal -> start_process_data -> tock -> move ->
        FUSION
                [] recieve_signal -> start_process_data -> tock -> tock ->
        move -> FUSION
                [] recieve_signal -> start_process_data -> tock -> tock ->
        end_process_data -> move -> FUSION
                [] recieve_signal -> start_process_data -> tock -> tock ->
        end_process_data -> start_fuse_data -> move -> FUSION
                [] recieve_signal -> start_process_data -> tock -> tock ->
        end_process_data -> start_fuse_data -> tock -> move -> FUSION
                [] recieve_signal -> start_process_data -> tock -> tock ->
        end_process_data -> start_fuse_data -> tock -> end_fuse_data ->
        move -> FUSION
                [] recieve_signal -> start_process_data -> tock -> tock ->
        end_process_data -> start_fuse_data -> tock -> end_fuse_data ->
        output_decision -> end_min_wk -> move -> FUSION
```

As can be seen from the models, after a potential wait, the process should receive a signal and then process the data received before fusing the data it has and data it has received. Once the component has fused the data it can make a decision based upon it and output this. At any point during this reconfiguration could occur from external or internal stimuli. Also if no data is received for 6 periods of time (3 mintes) it will output a decision based upon stale data and restart the same process. It was decided that it should keep refreshing the data after 3 minute intervals, as it could take an additional 1.5 minutes to process the data incoming if received later and it seemed logical to keep the cyclic time of the process to 5 minutes or less to be in keeping with the sensors, as well as allowing for some reconfiguration an still ensuring an output.

Just like the radar sensor, the data fusion and decision making component can reconfigure at any time triggered by external stimuli and as such the models have shown that it can suffer from configuration thrashing. The definition of configuration thrashing used within this model is 2 overlaps in 20 time intervals (10 minutes). This was used as the time taken to complete a successful cycle in the worst case is 4.5 minutes and thus if we have 2 overlaps inside the 5 minute deadline then there is a possibility that the deadline has not been met.

A restricted version of the data fusion and decision making process has been modelled. The restricted process has been modelled much like the restricted radar process and ensures that after a reconfiguration occurs, another does not until the minimum work is completed. Definitions have not been included in this chapter as this is very similar to the definitions for the

restricted radar process. However, the restricted process is contained in appendix E along with the complete case study model.

Section 3.4 discusses various difficulties in applying the configuration thrashing model. This case study has reinforced many of these issues. For example it was difficult even within such a small case study to decide upon the minimum level of processing needed before reconfiguration can occur. A single cycle of the process was decided upon, but this would have been made even more difficult if the processes involved were more complex. Also even with the minimum level of processing decided upon and deadlines being provided it became difficult to decide upon the values to place in the model for the number of overlaps in a given time period. In this case study it was made slightly easier because reconfiguration could occur at anytime, but even with this turning the deadlines into a number of overlaps that could occur in a time period was a little tricky.

The limitations introduced in section 3.3.3 also became apparent within this case study as the deadlines for the sensors really come from the deadlines in place over the data fusion and decision making component. The data needed to be sent from the sensors within 5 minutes to allow the data fusion and decision making component enough time to process the data, fuse the data and make a decision within its 10 minute deadline. However, the CSP models only allow single processes to be analysed and as such this type of dependency cannot be analysed.

Even though various difficulties and limitations became apparent within this case study, it also highlighted the usefulness of the models, as the processes were checked for configuration thrashing capabilities and where these capabilities were shown, they enabled for more restricted models to be developed that cannot thrash using the information gathered from the modelling process. Thus the models enabled configuration thrashing to be developed out of the processes.

6.3 Software Approach

Using the demonstrator introduced in section 5.2.8 this section introduces how the three processes described have been developed and shows how the configuration thrashing can be restricted. Discussions relating to the issues that had to be overcome both in terms of developing the implementations, and also in terms of using the restriction techniques are presented.

The demonstrator *ProcessFactory* and *Controller* processes were in the main unchanged for this case study. Each of the sensor processes and the data fusion and decision making process was implemented by extending the

current *ReconfigProcess* interface and a primary thread has been defined for each process. To simplify the implementation the *ReconfigProcess* interface was extended slightly allowing for public functions for the data fusion and decision making process to receive data through public functions without the use of the RMI stub narrow functionality. Also it was assumed that there would be only one data fusion and decision making component and that it would be bound as FusionDM which make the implementation a little simpler whilst still enabled the full level of reconfiguration functionality required.

As with the CSP models introduced in section 6.2 sensor data is simulated. All of the simulation is done via the use of a random number generator which if generated is an even number then it is treated as an input been received. If an input is received then the loop polling for input is broken and processing begins. The processing and fusion algorithms are not implemented, but instead a delay is used to simulate the worst case execution time of the algorithm. It should be noted that timings within the case study are ran at a tenth of the timings introduced in section 6.1 to enable rapid testing.

Within the case study component design it was assumed that data communications will always be reliable and instant or so close to it that it is negligible. However, the demonstrator uses true RMI communication and as such although was executed upon a single machine and thus should be reliable could have communication faults and these have to be coped with. Also communication is not instant but as the demonstrator is executed on a single machine it will be negligible.

The main functionality for all of the reconfigurable components is provided by the threads started as the service is initialised. The service simply allows for remote method invocation to occur and thus for statuses to be set or inputs / outputs made to or from remote objects.

The CSP models for the case study, shown in section 6.2, show that the radar sensor, the data fusion and decision making process can suffer from configuration thrashing when not restricted and the ground sensor cannot as it cannot reconfigure until the minimum level of work is completed. Testing of the processes developed using the demonstrator has shown this to be true. This type of testing using the demonstrator proved to be very interesting and showed exactly how the data fusion and decision making component rely upon the two sensors for input.

Three test classes have been created to test the data fusion and decision making component's reconfiguration and highlight its ability to suffer from configuration thrashing as well as show how rules can be used to restrict its

reconfiguration thus eliminating configuration thrashing. The first class, named *Start.java,* gets a reference to the controller object and creates the data fusion and decision making process, the radar sensor and the ground sensor. As none of the processes defined directly reconfigure them selves or each other with the exception of the ground sensor, and as such the system does not suffer from configuration thrashing unless external stimuli is provided. The ground sensor only reconfigures after a successful output is made which is deemed to be the end of its minimum work and as such cannot suffer from configuration thrashing.

The second class, named *Reconfigure.java*, provides external stimuli to reconfigure the data fusion and decision making process 30 times with a short time delay of just over 3 seconds between each reconfiguration. This causes the data fusion and decision making process to thrash thus further confirming what was shown in the CSP modelling. Outputs from the three processes using the *Start* and *Reconfigure* classes can be found in Appendix E subsection 3.1.

The third class, named *StartCont.java*, is very similar to the *Start* class. The main difference is that when the reference for controller object is returned, rules are enabled and a new rule is added to the controller. The rest of the class runs just as the *Start* class. When testing the Reconfiguration class upon the constrained process, configuration thrashing is eliminated. Outputs from the three processes using the *StartCont* and *Reconfigure* classes can be found in Appendix E subsection 3.2.

The data fusion and decision making process is the only process fully tested for configuration thrashing and had rules defined for them. This was sufficient for this case study as the radar sensor will suffer from configuration thrashing in much the same way that the data fusion and decision making process does and the rule defined for this will be much the same, but with a different minimum work and time frame defined. Though the minimum work and time frame variables will be calculated in much the same way as the processes are very similar in construction.

The rule used in the *StartCont* class is a *ConfOver* rule. As described in chapter 5, the *ConfOver* operator takes three variables: x, y and z. In this operator x is the maximum number of configuration overlaps which may occur, y is the interval in which the x configuration overlaps may occur, and z is the "sufficient interval" between reconfigurations to not be classified as a configuration overlap. In this example x is set to 2, y is set to 100 and z is set to 50 in order to eliminate configuration thrashing. It should be noted that the all timings are set to be a tenth of the real timings, to allow for rapid testing.

As with the difficulties in applying the configuration thrashing model it was difficult even within such a small case study to decide upon the minimum level of processing time needed before reconfiguration can occur. Also even with the minimum level of processing decided upon and deadlines being provided it became difficult to decide upon the values to place in the *ConfOver* operator. Similar values have been used to that in the CSP models. The rule that $y <= (x+1)*z$ did assist in the decision and checking that it was correct. A big benefit of the using the controller is that testing can easily be conducted on the live system as it runs to check if deadlines are met with various values in place.

Often deadlines are discovered during system testing and as such CSP modelling would have to be amended to put the new deadlines in place. A distinct advantage of using the controller is that new rules can simply be added to the system or existing rules can be amended to suit the new rules. It would be hoped that as rules can be changed with relative ease, then the system rules put in place would not be over restrictive and altered as and when needed. Also as a system evolves, then each component can have its own unique set of rules that go into place to enforce its rules.

The software solution has highlighted that similar issues to the ones experienced with the CSP models are apparent, but also shown that configuration thrashing can very successfully be eliminated using a controller. The biggest benefit of the controller process is that the developer does not have to alter his or her process when requirements change, as rules can simply be altered or added instead.

6.4 Case Study Discussion

The two approaches to solving configuration thrashing used within this case study have both shown themselves to be successful and assist developers in either engineering configuration thrashing out of their systems or restricting it to the extent that it is no longer an issue.

The CSP approach provides benefits in that the model is validated to definitely not suffer from configuration thrashing under all circumstances. Whereas the software approach can be tested extensively and still there may be a chance that if the rules have not been thought through fully, then the system could thrash. However, modelling has to be done over and above the software development to use the CSP configuration thrashing models to check processes for configuration thrashing which developers may see as over restrictive.

This case study has shown that the software approach is far more flexible than the formal CSP approach as rules can be gradually changed to suit new and emerging requirements. When requirements were tweaked during the process of creating the CSP models used within this case study a sizable amount of rework was required, whereas the software approach would simply need an amendment to a rule or a new rule.

The software approach allows for an evaluation of many processes simultaneously and enables the reviewing of interactions. The CSP approach has a limitation in that it can only review single processes presently though this has been detailed as a future improvement.

As stated within this thesis modelling can suffer from state space explosion and as such complex models may not be able to be evaluated for configuration thrashing using the formal approach detailed in this thesis. As well as this one of the common issues with modelling is that if models do not reflect the system perfectly then all that the assertions prove is that the model cannot thrash and does not show anything regarding the actual implementation.

Neither of the approaches really assisted in choosing how to define the minimum level of work that must be completed before a reconfiguration occurs, or how many overlaps can occur in a given time period. Experimentation can be conducted with both approaches, but since the CSP models as they stand only assess configuration thrashing and not the ability to meet deadlines, the software approach lends itself to this a little more.

6.5 Summary

This chapter has introduced a case study which has been used to show the usefulness of both the CSP modelling approach to eliminating configuration thrashing and also the software controller based approach to restricting systems reconfiguration actions based upon rule based logic. The case study has clearly highlighted some weaknesses and difficulties in using both of these methods, but also highlighted that both approaches can be used to eliminate configuration thrashing successfully and each have many unique benefits.

Both of the approaches have been compared and contrasted. The CSP approach has a huge benefit in that if the system is modelled accurately then the system can be proven to not be capable of configuration thrashing, whereas the software approach cannot provide these guarantees. However the software approach is far more flexible and offers developers an approach which fits the development methods that are already employed.

Chapter 7

Future Work

This thesis has presented a definition of configuration thrashing and explored various methods of eliminating it from reconfigurable systems. A formal approach has been presented using model checking techniques (CSP and FDR), which allows configuration thrashing to be engineered out of processes. Run-time techniques have also been explored, allowing developers to include additional logic / processes in their systems in order to prevent configuration thrashing. This chapter presents areas of future work, some of which are extensions to work presented in this thesis, and others that are related work not within the scope of the thesis. All future work proposals presented in this chapter would contribute towards certifiable dynamic reconfigurable systems capable of meeting deadlines.

The rest of this chapter is structured as follows. First, section 7.1 describes future work in the form of blueprint to blueprint reconfigurations and the analysis of configuration thrashing upon groups of processes. Some discussions relating to options for blueprint representations and potential solutions to allowing blueprint to blueprint reconfiguration to occur are introduced. Section 7.2 considers future work on resource modelling and methods of proving equivalence of given resources; if achieved this work would allow software to be reconfigured at run-time in the presence of diverse hardware without the need for additional certification. Section 7.3 describes work relating to contract restrictions for middleware systems. This approach could potentially allow message oriented middleware to be used in real-time reconfigurable systems. Section 7.4 discusses the work that would be needed to allow dynamic rule sets to be used in reconfiguration control systems, thus allowing potential benefits such as run-time upgrades.

7.1 Blueprint to Blueprint Analysis

The definition for configuration thrashing presented within this thesis focuses upon a single process and as such the CSP models presented within this thesis are also focussed upon a detecting configuration thrashing in single processes.

As discussed in section 3.3.3, complex interactions will most likely exist between processes therefore developers are likely to consider

reconfiguration as a step from one system layout (or blueprint) to another. This is especially likely if the reconfiguration is intended to change mode.

All of the models and definitions within this thesis allow configuration thrashing to be detected, removed or restrained on a process by process basis. Although, none of the models or definitions consider the fact that interactions between groups of processes may mean the processes that in theory can "thrash" (and would be flagged as needing alterations in the current CSP models) cannot when the group of interacting processes cannot produce the stimuli required to trigger the configuration thrashing. Thus further research is required to take into account the complex interactions that exist between processes. In order to analyse the interactions it is important that we also investigate how a system can step from one blueprint to another blueprint as this is the process in which configuration thrashing will occur. In order to do this we also must define what a blueprint is.

Research in IMA / IMS has attempted to map system layouts using blueprints [71, 72]. However, blueprints are not precisely defined in any papers found to date. A distinction has been made between Design Time Blueprints (DTBPs) and Run Time Blueprints (RTBPs). Although, the distinction between RTBPs and DTBPs is entirely based upon usage, i.e. whether or not they are used in a live system.

In [1] RTBPs are defined as "[t]he mapping of which part of which application goes onto which hardware module in the IMA system" which is a reasonable definition, albeit vague. While discussing blueprints in [72] the author states "[t]he generation of the blueprints depends on accurate information about the applications, (eg memory, processing, timing requirements) and about the hardware in the system, (eg memory and processing availability)", which indicates that precise knowledge of the system hardware must be available in order to design blueprints, but this does undermine certain aims of IMA such as plug and play hardware.

Bradley et al [71] state that "[b]lueprints provide the generic operating system with configuration information so that the MAOS can be adapted for a particular avionics system", which is no more descriptive than the definition found in [72]. However, Bradley et al go on to state that blueprints include:

- The application run-time requirements
- Allocation and scheduling tables and rules (generated by the off line allocation and scheduling tools)
- Hardware resources descriptions

In the authors opinion three possible methods of representing blueprints are available. Each of these is briefly described below:

1. **Static Hardware Mappings** – this would involve mapping individual processes to individually selected physical hardware.
2. **Resource Mappings** – this would involve mapping processes to resource requirements (i.e. processor and memory requirements). The system would map the processes onto hardware when the blueprint is activated. This may require certain elements of the future work presented in section 6.2 to allow resource equivalence to be reasoned about.
3. **Hybrid** – this would physically map critical processes onto individually selected physical hardware, but use resource mappings for the rest of the system. It is assumed that in this type of blueprint the critical processes in the blueprint would not vary significantly between blueprints.

In the authors opinion resource mappings provides the most promising option as it provides the most flexibility and allows increased dynamisism within systems. If used could be constructed as a combination of resource requirements and communication patterns. Communication patterns would specify the interactions that should occur between processes. Both the communication and resource elements are required since much of the resources required for processes are dependent on the communication resources they require.

There are likely to be many possible valid hardware mappings for each blueprint. Figure 1 shows the relation between modes, blueprints and hardware mappings:

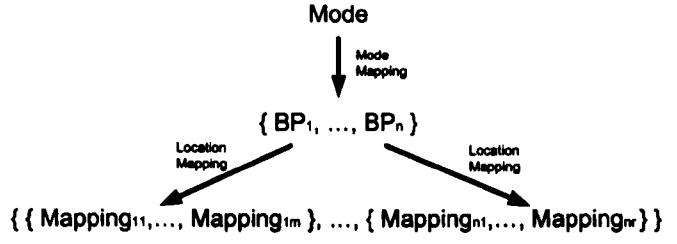

Figure 1: Mode, Blueprint (BP), and Mapping relationship

To allow reconfiguration between blueprints there must be a method of taking a blueprint in a given format and storing it in the system. Then when the blueprint is required to become active, some part of the system must calculate a valid process to hardware mapping and reconfigure the system to the valid configuration. It should be noted that a process should be in place

to ensure that system invariants are not broken during this reconfiguration. This could in itself form a PhD as formally proving that dynamic large scale system changes can never break invariants will be very challenging, particularly if we are to introduce fault handling during the same process.

The number of transitions involved in a reconfiguration will require investigation as it is likely that there will be many sets of transitions which could take a system from its initial configuration to the valid reconfigured configuration. Establishing the best set of transitions to use is an issue to be considered for further research and a minimalist approach to this seems sensible as the less transitions and process movement involved, the easier the enforcement of hard real-time deadlines will be.

Once questions over how blueprint to blueprint reconfiguration will occur have been answered, then it will be possible to extend this research to consider configuration thrashing in groups of processes that reconfigure using blueprints. This should allow more flexibility to developers when considering configuration thrashing.

7.2 Resource Modelling / Equivalence

In many systems diverse hardware exists, but in most of these systems the software is only tested and certified for the hardware it was intended to execute upon. If hardware is altered, re-certification is required and is often a lengthy and costly process. Re-certification must be conducted to ensure that the applications can meet necessary deadlines upon the new hardware. In fully dynamic reconfigurable systems both hardware and software could reconfigure; thus software could end up attempting to run on hardware that it was not originally intended to execute upon.

In order to support reconfigurable systems which adapt themselves in the presence of diverse hardware, it is necessary to dynamically assess if a process can execute and meet deadlines on hardware it was not originally intended to execute upon. One way to achieve this assessment is to model hardware resources and process requirements, thus allowing checks to be conducted before reconfiguration takes place. This should also allow equivalences between hardware to be analysed. This may seem to be a relatively straightforward task; however, hardware resource modelling and process resource requirement modelling are not trivial. Hardware resources have many attributes and each attribute is equally important. Processor attributes may include the following:

- Cpu speed (e.g. 1200MHz)
- Cache size
    - L1 cache size
    - L2 cache size
    - L3 cache size
- Pipelining
- Operating voltage
- Bus frequency (MHz)
- Number of channels
- Core frequency (MHz)
- Bus/Core ratio
- Instruction sets (e.g. 3D NOW)
- Co-processor
- Register size (e.g. 32bit, 64bit – particularly makes a difference to the amount of physical memory which can be accessed)
- Cycle time

Appendix C contains a more comprehensive list of possible processor, memory, OS, and storage attributes. The attributes outlined in appendix C are only a candidate set of attributes and many others could be defined.

Each individual attribute could have a large impact upon executing processes. For instance pipelining attempts to improve the performance of processors in large sequential programs; however, in some cases the pipelining decreases performance. Thus pipelining in general improves performance, but when conducting worst case analysis for program execution, it makes it worse. Also attributes cannot be assessed independently. For example it does not necessarily hold that a process which can meet its deadlines on a 1200Mhz processor can meet its deadlines on a 2000Mhz processor, as it could have a different instruction set or use an inefficient instruction. A partial match is unlikely to be sufficient.

Further work is required to find a suitable method of expressing resources and also resource requirements. The expression of resources and requirements must be impossible to misinterpret and also simple to analyse. Also an algorithm for equating resources and requirements is required. This algorithm is likely to need a minimum number of attributes and requirements in order to provide a match, though further work is required to state what the minimum set would be.

If process requirements are expressed accurately and a timely algorithm can be developed to match hardware attributes to software requirements, then it

is possible that software could be reconfigured at runtime in real-time applications where diverse hardware exists with no need for recertification.

7.3 Contract Restriction for Reconfigurable Middleware Systems

Middleware supports communication between distributed objects, abstracting away the networking issues from the application designers / developers. An important aspect of middleware systems is location transparency, allowing clients to remain unaware of the location of a component or service. Location transparency naturally lends itself to reconfigurable systems, since when a service moves the client does not need to be aware of the move. Message Oriented Middleware (MOM) lends itself particularly well to reconfigurable systems as it not only allows for location transparency, but also allows asynchronous communication and provides support for multi-casting. MOM can distribute the same message to multiple clients in a way which is transparent to the clients.

Publish-subscribe systems are an advanced type of Message Oriented Middleware. In a publish-subscribe system a message sender does not specify the address of any receiver. Instead, the sender publishes an event with a subject (filter), while the receivers who are subscribing to the subject will receive an asynchronous notification event. This provides a mechanism which could allow reconfigurable components to register for message receipt upon reconfiguring instead of informing the message publisher of the reconfiguration. Publish-subscribe systems also allow for complete decoupling of source and target. In fact in [73] it is argued that publish-subscribe systems allow for decoupling of source and target along three dimensions; space, time and flow. These are described below:

- **Space decoupling:** the publisher does not know who the subscribers are (if any) and the subscribers do not know who the publisher is. This means that the publisher has no reference to the subscribers.

- **Time decoupling:** the interacting parties do not have to be actively participating in the interaction at the same time, and so the publisher may publish an event when a subscriber is offline; the subscriber will be notified of the event once it comes online again (though the publisher may or may not be online at the same time).

- **Flow decoupling:** the flow of the messages from publisher to subscriber is not synchronised upon by the publisher or subscriber. There is no blocking or polling by either party.

This level of decoupling allows for a highly reconfigurable system. Within a publish-subscribe system if a process wishes to receive messages, it can subscribe and a route will become available, provided that hardware connectivity is available, allowing messages to be received by the new subscriber; the route does not have to be pre-defined.

Though the 'publish-subscribe' paradigm has many benefits, the publish-subscribe paradigm (in its present form) is not suitable for real-time systems as it is not temporally predictable, and the resource requirements are not predictable. In [63] T. Murata & N.H. Minsky have stated that the publish-subscribe paradigm "... has a dark side, which may complicate the system using it, making it less predictable, more brittle, and less safe", and they have gone on to suggest that restrictions can be placed on the publishers and subscribers to alleviate the 'dark side'.

The benefits provided by publish-subscribe system have not gone unrecognised, as publish-subscribe systems have been used in real-time systems, in particular systems for the US Homeland Defence [74]. However, even in this work the issues relating to temporal predictability have not been addressed. It is believed that contracts could be used to restrict communication and ensure temporal predictability thus allowing it be used in real-time reconfigurable systems.

Further work should be conducted to investigate the possibilities for restricting the publish-subscribe paradigm using contracts. It is proposed that the use of intermediaries ("brokers") between a publish-subscribe system and the publishers/subscribes could be used to enforce contracts. However, it must be noted that brokers need to be simple and predictable to allow accurate timing information to be gathered. Timing information is required for brokers, in order to ensure deadlines will be met. An investigation should be conducted to establish which properties of a contract can be enforced locally (within a single broker), which require interaction with other brokers, and which require interaction with the publish-subscribe system itself (this may include properties which require modifications to the publish-subscribe system to allow for enforcement).

If middleware systems such as the publish-subscribe paradigm can be restricted, then as indicated above, they could present a flexible architecture that would enable the development of highly reconfigurable real-time systems.

7.4 Dynamic Rule Sets for Reconfiguration Control

Chapter 5 explored potential run-time solutions to configuration thrashing, a rule based solution was explored in-depth and various issues were discussed. Chapter 5 also introduced the notion of dynamic rule sets, whereby rule sets can be altered at runtime. Dynamic rule sets could be used to control configuration thrashing by imposing rules upon a process reconfiguration to ensure that the reconfigured processes remain static until a new process reconfiguration request would not constitute configuration thrashing. This is a novel solution to restricting configuration thrashing and could potentially take less computational resources than the other run-time approaches discussed within the thesis.

The use of dynamic rule sets would affect the temporal predictability of the controller's decision making, as the number of rules that exist within the system will vary; rules added to stop reconfiguration occurring and thus stop configuration thrashing should be removed once they become stale. Removal of rules would incur a processing overhead, which could alter the worst case execution time for the decision making algorithm. It is possible for the stale rules to remain within the system, as once the time period is exceeded the rules will have no effect. However, the effect on the worst case execution time of leaving all stale rules in place would most likely be worse than the overhead required removing the stale rules.

Additional benefits such as quicker upgrades and run-time alterations could be provided if dynamic rule sets were to be used. It is possible in some cases that additional rules would be required for specific new components upon there inclusion in a system - these new rules could be added at run-time provided they do not contradict existing rules.

Further work is required to further explore dynamic rule sets. A temporally predictable run-time consistency checking algorithm will require further research. Also a method of ensuring that new rules added at run-time do not contradict existing rules is required. This thesis did not further explore dynamic rule sets, as they could themselves become susceptible to a form of configuration thrashing. If a temporally predictable run-time consistency checking algorithm can be developed for rule sets then further research in this area would certainly be worth progressing.

7.5 Summary

This chapter introduces four avenues for future work. The future work presented is either extensions to work presented, or related work that is not within the scope of the thesis.

The first piece of potential future work introduced the difficulties in resource modelling and methods of proving equivalence of between resources. This work could allow software to be reconfigured at run-time in the presence of diverse hardware with no need for additional certification. The second piece of potential future work describes work relating to contract restrictions for middleware systems. This could potentially allow message oriented middleware to be used in real-time reconfigurable systems. The third piece of potential future work would allow dynamic rule sets to be used in reconfiguration control systems, which could provide benefits such as run-time upgrades.

Chapter 8

Conclusions

This thesis has introduced reconfigurable systems, and identified an anomaly that can occur within reconfigurable systems whereby a system consumes most, if not all, of its resources reconfiguring and thus cannot execute intended computing functions. This has been named "configuration thrashing" due to its similarities to memory thrashing. The main objectives for this work were: first, to introduce reconfigurable systems and explore the possibilities within reconfigurable systems to enable an unambiguous extensible reconfiguration language to be developed; second, to characterise and define configuration thrashing and investigate the effects it has upon real-time reconfigurable systems; and third, to develop methods by which the effects of configuration thrashing on reconfigurable real-time applications can be eliminated or at least reduced sufficiently to stop reconfiguration from interfering with intended computing functions.

Model checking is utilised within this thesis to provide a means of ensuring that configuration thrashing is engineered out of systems. Potential run-time solutions to configuration thrashing are also explored. Run-time solutions are explored because model checkers are not adequate for large complex systems, as these will suffer from state space explosion. The work presented in this thesis provides a step towards certifiable dynamic reconfigurable systems capable of enforcing deadlines. The elimination of configuration thrashing is necessary, though not sufficient, for this goal.

The remainder of this chapter is organised as follows. First, Section 8.1 briefly summarises the exploration of possibilities within reconfigurable systems using a VDM-SL model. Section 8.2 discusses the definition of configuration thrashing. Section 8.2.1 discusses the elimination of configuration thrashing using model checkers, and Section 8.2.2 looks at run-time techniques for configuration thrashing elimination. Section 8.2.3 reviews the effectiveness of the two approaches introduced to eliminate configuration thrashing by drawing upon the case study introduced in chapter 6. Section 8.3 gives the final conclusions.

8.1 Reconfigurable Systems

Reconfigurable systems offer the ability to adapt hardware and / or software to meet changing requirements. Reconfigurable devices, particularly Field-Programmable Gate Arrays (FPGAs) have been the subject of increased

popularity, due to having been shown to accelerate a number of computing applications.

Reconfigurable software provides the ability to alter software systems either in terms of software linkage, or adding / removing processes. Online reconfigurable software systems offer many potential benefits over systems only capable of offline reconfiguration including: online software upgrades, adaptability, self-management, and increased fault-tolerance.

Research into reconfigurable systems has shown that a suitable language in which to express the behaviour of reconfigurable systems is lacking. Many terms used in reconfigurable systems research are not well defined and thus can be confusing or even ambiguous. Chapter 2 introduced a three-level model which has been specified formally in VDM-SL to provide a basis for exploring the possibilities available within reconfigurable systems. The VDM-SL model has been built to allow architectures to be manipulated using a set of well defined reconfiguration operators. The operators outlined form an unambiguous extensible reconfiguration language. The VDM-SL model is very detailed and all operators specify implementation detail, thus eliminating ambiguity.

The VDM-SL model has yielded many interesting insights. It has shown that the number of options available within reconfigurable systems is greater than anticipated. The model has also shown that although proxies are not commonly associated with reconfigurable systems, they may have a valuable role to play in dynamic reconfigurable systems, when a totally interconnected network is not available. Proxies also offer the benefit of buffering messages whilst reconfiguration occurs.

As well as providing insights into the options available for reconfigurable systems, and how individual reconfiguration operators can be implemented, the model has also provided interesting insights into reconfigurable architectures. For example, it has shown that it seems necessary to allocate all processes a global unique identifier to avoid reliance on (inadequate) location-dependent references, as well as showing that even with minimal system invariants, some operations require atomic actions to ensure system invariants are not violated during reconfiguration.

The research into reconfiguration operators has shown that the implementation of the operators specified in the VDM-SL model would in the main be difficult, though not impossible. Certain operators may require OS support, for example operators which synchronise processes' instruction stacks may require OS support to write to such private memory areas.

8.2 Configuration Thrashing

Configuration thrashing is an anomaly which can arise in real-time reconfigurable systems. It is, in essence, a lack of progress of intended computing functions due to reconfiguration activity consuming essential resources, thus causing deadlines to be missed.

It may be argued that deadlines are missed and progress is not made in reconfigurable systems due to non-reconfiguration functionality being inefficient, rather than reconfiguration actions utilising required resources. The author does recognise that in some cases improvements in non-reconfigurable actions could allow processes to meet deadlines without reconfiguration alterations, though it is not possible in all cases. In extreme cases reconfiguration could take place continuously, thus making it impossible for non-reconfiguration functionality to make progress no matter how efficient it is.

Configuration thrashing is defined within Chapter 3 of this thesis as "...occurring when one or more configuration overlaps occur. The number of configuration overlaps that can be tolerated in a given time period or in a given sequence is application dependent and possibly even mode dependent...". A configuration overlap occurs when two subsequent reconfiguration requests are executed without a "sufficient interval" between them. The sufficient interval required in a given configuration is application dependent. The worst case scenario is a never ending series of consecutive configuration overlaps, which logically will always be classified as configuration thrashing as progress cannot be made.

If configuration thrashing is not eliminated then it is possible for a situation to arise where a reconfigurable system cannot provide sufficient resources to conduct its primary computing functions due to reconfiguration actions utilising required resources.

Though no literature found to date specifically explores configuration thrashing. Related work in the areas of *fault-tolerance*, *reflection*, *self modifying code* and *re-configurability* in general have been explored in an attempt to put configuration thrashing in context with similar problems found in these related research topics. This exploration found that many pieces of similar work can suffer from configuration thrashing, but very few actually recognise that this is indeed an issue. The few that do recognise timing issues as a problem have reviewed this in terms of quality of service and utilise feedback based algorithms which are not adequate for hard real-time systems. Also very few pieces of related work have a solid formal underpinning and the few that do focus upon safety through proving system invariants hold throughout reconfiguration, however this and other similar

research does not address timing issues and as such does not and cannot address any issues similar to or relating to configuration thrashing.

8.2.1 Eliminating Configuration Thrashing Using Model Checkers

Un-timed and timed CSP models capable of detecting the possibility of configuration thrashing are presented in Chapter 3. The un-timed CSP model allows configuration thrashing to be defined in terms of a sequence of consecutive configuration overlaps. However, this may not be adequate, as it may be required that configuration thrashing be defined as x overlaps in a given time period. The timed CSP model allows this more general definition of configuration thrashing to be used.

CSP in its traditional form has no notion of time, though there are two distinct approaches to expressing time in CSP. The more elegant is to re-interpret the CSP language to log the time for each event which occurs. The alternative approach is a discrete model of time, which makes the drum-beat of time an explicit event. The interval between successive "beats" may be any finite duration. The discrete model of time was adopted by the author within the timed CSP model as although the continuous approach is more elegant, the discrete approach offered the level of tool support required for experimentation.

The specification of the timed CSP model has shown that data freshness is important, because in order to detect configuration thrashing an event history must be maintained and events in this history will become stale. This was not required within the non-timed model as without time data cannot become stale.

Both the timed and un-timed CSP models can be used to ensure that configuration thrashing is engineered out of systems. However, it was found that in some cases it may be impractical or even impossible to use model checkers, as model checkers (such as FDR) are susceptible to state space explosion [19]. This is particularly true of large complex system models, though smaller less complex system models should not suffer unduly and many techniques can be used to limit the effects of state space explosion during modelling.

Experimentation with the CSP models has led to interesting findings relating to difficulties in applying the CSP models. For example, probabilistic requirements are often imposed on systems, which are challenging for system developers, as a deadline will only be valid for a proportion of the system's operational activity. As configuration thrashing

is effectively reconfiguration causing deadlines to be missed, configuration thrashing itself can also become probabilistic. In many cases developers make deadlines over probabilistic requirements hard, in order to alleviate complications, however this leads to over-engineering, and in some cases these over engineered requirements can conflict unnecessarily with normal (non over-engineered) requirements. Further discussions regarding difficulties in applying the configuration thrashing models are presented in Chapter 3.

8.2.2 Run-time Techniques for Configuration Thrashing Elimination

Potential run-time solutions to the problem of configuration thrashing are explored in Chapter 5. These solutions allow developers to include additional logic / processes in their systems in order to eliminate configuration thrashing. Several options are explored in-depth, from providing mechanisms that enable developers to choose when reconfiguration can / cannot occur, to a more automated rule based solution.

Many methods of allowing developers to choose when reconfiguration can / cannot occur have been explored. Providing engineers with reconfiguration mechanisms in the form of a service is one example. Methods such as this would allow developers direct control over when and how reconfiguration could occur on a process by process basis. This provides a novel approach, but as discussed in chapter 5, without guidance developers may not know when reconfiguration should take place or, more importantly, when it would be "safe" for it to take place. Developers may be tempted to develop systems that are more static than necessary.

Also investigated within this thesis is a rule based solution, in which a reconfiguration controller sub-system decides when reconfiguration can and cannot occur, based upon logic defined as a set of rules. Traditionally logic within control processes is hard coded; however, benefits such as quicker upgrades and run-time alterations could be provided if the logic in a reconfiguration controller is specified as a set of alterable rules. This type of approach allows system developers to focus on core development without concern for reconfiguration issues. The rule based approach has highlighted many interesting issues, such as rule expression, rule predictability, as well as potential core rules for systems; all of these issues are further discussed within Chapter 5.

A rule based demonstrator has been developed. The demonstrator provides a basis for reviewing the effects of configuration thrashing on a real-time system. Experimentation has been conducted using the demonstrator, not

only for rule sets but also for scenarios where developers control reconfiguration, as it was constructed to allow reconfiguration operators to be accessed directly.

The demonstrator has clearly shown that the candidate operators specified within the VDM-SL model can be implemented (though not all of the operators were actually implemented). The demonstrator has also enabled experimentation to be conducted using restriction rules. An example process has been developed and many rules were tested against this process; the outcome was that reconfiguration could be restricted sufficiently within the demonstrator to eliminate configuration thrashing. Details relating to the experimentation conducted are considered further in Chapter 5.

8.2.3 Configuration Thrashing Elimination Effectiveness

To review the effectiveness of both the models outlined for configuration thrashing, and also the run-time solution outlined, a small case study has been developed. This case study is focussed upon battlefield surveillance using multi sensor data fusion. The ability to rapidly detect and identify potential targets both fixed and mobile from multiple sensor inputs is a critical function in modern warfare.

Within a battlefield surveillance system targets need to be assessed as quickly as possible in order to guide troops accurately and ensure that weapons are do not target non hostile targets. Within the case study two different types of sensor are outlined, as well as the main data fusion and decision making process.

Both the formal CSP approach and the run-time rule based approach enabled configuration thrashing to be eliminated. The CSP approach enabled for a restricted model to be constructed that could not reconfigure and the run-time rule based software approach enabled for reconfiguration to be restricted sufficiently to avoid configuration thrashing. Both approaches have their own limitations, but both are very effective.

This case study has shown that the run-time rule based software approach is far more flexible than the formal CSP approach as rules can be gradually changed to suit new and emerging requirements. When requirements were tweaked during the process of creating the CSP models a sizable amount of rework was required, whereas the software approach simply required a small rule amendment.

The CSP approach has a limitation discussed in chapter 3 which is that it can only allow configuration thrashing to be detected in single processes

and as such does not consider the reconfiguration of groups of processes. In distributed systems complex interactions will exist between processes and as such developers are likely to consider reconfiguration as many processes reconfiguring simultaneously or in a well defined sequence. The models produced can check if each individual process can "thrash" and as such be used to check entire systems (one process at a time), but this does not consider the fact that interactions between the groups of processes may make the processes that in theory can "thrash" not capable of configuration thrashing as the interacting processes may not provide the necessary stimuli. This has been highlighted as an area of future work. Though this was not reinforced within the case study, it was in many ways easier to work with the run-time rule based approach as you could see and interpret the interactions between the processes more easily.

Neither of the approaches really assisted in choosing how to define the minimum level of work that must be completed before a reconfiguration occurs, or how many overlaps can occur in a given time period before configuration thrashing occurs. However, experimentation can be conducted within both approaches, but since the CSP models as they stand only assess configuration thrashing and not the ability to meet deadlines, the run-time rule based software approach lends itself to this better.

Although many limitations and difficulties were highlighted within the case study, it also showed that both approaches could be used to eliminate configuration thrashing and did so very effectively within the case study.

8.3 Concluding Remarks

The contribution of the work in this thesis is firstly, the development of a VDM-SL model allowing the behaviours of reconfigurable systems to be expressed, as well as outlining a set of operators which form an unambiguous extensible reconfiguration language that can be used in system development. Secondly, an anomaly termed configuration thrashing has been explored in detail and a formal definition has been presented.

Related work in the areas of *fault-tolerance, reflection, self modifying code* and *re-configurability* in general have been explored in an attempt to put configuration thrashing in context with similar problems found in these related research topics. Very few pieces of related work recognise that this is indeed an issue, but most can suffer from it. The few that do recognise similar issues review them in terms of quality of service and are not adequate for hard real-time systems. Very little of the related work reviewed had a solid formal underpinning and the few that did focus upon

safety invariants rather than addressing timing issues and as such cannot address any issues similar to or relating to configuration thrashing.

Building on the configuration thrashing definition, a further contribution is made in this thesis by exploring methods which can be used to eliminate configuration thrashing in reconfigurable systems. A formal approach to the elimination of configuration thrashing has been presented using model checking techniques (CSP and FDR), and a range of run-time techniques have been explored.

A contribution has also been made in the form of a demonstrator which has allowed a level of experimentation to be conducted to further demonstrate that the range of run-time techniques can indeed restrict configuration thrashing sufficiently. The demonstrator has also shown that in all attempted cases the candidate set of operators defined within the VDM-SL model can be implemented.

The case study presented within this thesis also further highlights the usefulness of both the CSP models and also the run-time rule based software approach to restricting reconfiguration to eliminate configuration thrashing. Both of the approaches were shown to allow configuration thrashing to be eliminated in the battlefield surveillance case study presented. The case study did further confirm many of the limitations and difficulties discussed within the thesis when applying the models or attempting to decide upon appropriate rules for the run-time rule based solution. Many of these limitations provide the basis for future work presented in chapter 7 of this thesis. However, none of these limitation stop configuration thrashing from being eliminated, but in some cases could lead to the system being unnecessarily over restrictive.

The work presented in this thesis provides a step towards certifiable dynamic reconfigurable systems capable of enforcing deadlines by investigating and providing methods of eliminating configuration thrashing.

Bibliography

1. A.J. Elbirtm, C. Paar, *An Implementation and Performance Evaluation of the Serpent Block Cipher*, in *ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays*. 2000. p. 33-40.

2. K. Compton, S Hauck, *Reconfigurable Computing: A Survey of Systems and Software*. ACM Computing Surveys, 2002. **34**(June 2002): p. 171-210.

3. Oreizy, P., *Issues in Modelling and Analyzing Dynamic Software Architectures*, in *Proceedings of the International Workshop of the Role of Software Architecture in Testing and Analysis*. June 30 - July 3 1998: Marsala, Sicily, Italy.

4. C Szyperski, *Component Technology: What, Where and How?*, in *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*. 2003. p. 684-693.

5. J.S.Bradbury, J.R.Cordy, J.Dingel, M.Wermelinger. *A Survey of Self-Management in Dynamic Software Architecture Specifications*. in *Proceedings of the International Workshop on Self-Managed Systems (WOSS'04)*. October/November 2004.

6. *Avionics Application Software Standard Interface*. January 1997, ARINC Specification 653.

7. M Nicholson, P Conmy, I Bate, J McDermid, *Generating and Maintaining a Safety Argument for Integrated Modular Systems*, in *5th Australian Workshop on Safety Critical Systems and Software*. 2000: Australia. p. 31-41.

8. R Milner, *Communicating and Mobile Systems: The Pi Calculus*. 31st May 1999: Cambridge University Press.

9. L Cardelli, A D Gordon, *Mobile Ambients*. Theoretical Computer Science, Special Issue on Coordination, June 2000. **240/1**: p. 177-213.

10. H Simpson, *Protocol for Process Interaction*, in *IEE Proceedings for Computers and Digital Techniques*. 2001. p. 157-182.

11. I Haynes, C B Jones, *Specifications are not (Necessarily) Executable.* Software Engineering Journal, November 1989. **4**(6): p. 330-338.

12. N E Fuchs, *Specifications are (Preferably) Executable.* Software Engineering Journal, September 1992. **7**(5): p. 323-334.

13. *DCE 1.1: Remote Proceedure Call.* August 1997 [cited; Available from: http://www.opengroup.org/publications/catalogue/c706.htm.

14. C A R Hoare, *Communicating Sequential Processes.* Prentice Hall International Series on Computer Science. 1985: Prentice Hall.

15. A W Roscoe, *The Theory and Practice of Concurrency.* 1998: Prentice Hall.

16. R Milner, *Communication and Concurrency.* Prentice Hall International Series on Computer Science. 1989: Prentice Hall.

17. Welch, P.H. *A CSP model for Java threads.* 1999 [cited 2008 10/11/2008]; Available from: http://www.cs.kent.ac.uk/projects/ofa/java-threads/203.html.

18. S Schneider, *Concurrency and Real-Time Systems - The CSP Approach.* 2000: Wiley.

19. W Reisig, G Rozenberg, *The State Space Explosion Problem.* Lecture Notes in Computer Science, 1998. **1491: Lectures on Petri Nets I: Basic Models**: p. 429-528.

20. A Burns, A Wellings, *Real-Time Systems and Programming Languages.* 3 ed. 2000: Addison Wesley.

21. H Hansson, B Jonsson. *A Calculus for Communicating Systems with Time and Probabilities* in *Proceedings of the 11th Real-Time Systems Symposium.* December 1990. Lake Buenna Vista, Florida, USA.

22. R Milner, *The Polyadic Pi-Calculus: A Tutorial in Logic and Algebra of Specification.* 1993: Springer-Verlag.

23. L Cardelli, A D Gordon, *Mobile Ambients, foundations of Software Science and Computational Structures.* Lecture Notes in Computer Science, 1998. **1378**: p. 140-155.

24. M Wermedlinger, J L Fiadeiro, *Connectors for Mobile Processes.* IEEE Transactions on Software Engineering, May 1998. **24**(5).

25. G Roman, P J McCann, J Y Plun, *Mobile UNITY: Reasoning and Specification in Mobile Computing.* ACM Transactions on Software Engineering and Methodology, July 1997. **6**(3): p. 250-282.

26. A Lopes, J L Fiadeiro, M Wermedlinger, *Architectural Primitives for Distribution and Mobility.* Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, November 2002: p. 41-50.

27. B C Pierce, *Basic Catagory Theory for Computer Scientists.* 1991: MIT Press.

28. Pierce, W.H., *Failure-Tolerant Computer Design.* 1965: New York: Academic.

29. J J Horning, H.C.L., P M Melliar-Smith, B Randell *A program structure for error detection and recovery*, in *Lecture Notes In Computer Science; Vol. 16, Operating Systems, Proceedings of an International Symposium.* 1974, Springer-Verlag: London, UK. p. 171 - 187.

30. Randell, B., *System Structure for Software Fault Tolerance.* IEEE Transactions on Software Engineering, 1975. **1**: p. 220--232.

31. B Randell, A.R., R J Stroud, J Xu, A F Zorzo, ***Coordinated Atomic Actions: from Concept to Implementation.*** Submission to IEEE TC Special Issue 1 1, 1997.

32. P A Lee, T Anderson, *Fault Tolerance: Principles and Proactive.* 2nd Edition ed. 1990: Springer-Verlag Wien New York.

33. S Porcarelli, M.C., F Di Gi, A Bondavalli, P Inverardi *An Approach to Manage Reconfiguration in Fault Tolerant Distributed Systems.* in *Proceedings of the ICSE 2003 Workshop on Software Architectures for Dependable Systems* 2003. Portland, Oregon, USA.

34. J Fraga, F.S., F Favarim *An adaptive fault-tolerant component model* in *Proceedings of the Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03).* 2003. Capri Island: IEEE.

35. G. Coulson, G.S.B., M. Clarke, N. Parlavantzas, *The design of a configurable and reconfigurable middleware platform.* Distributed Computing, 2002. **15**.

36. Y. Sizhong, L.J. *RECOM: A Reflective Architecture of Middleware.* in *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns.* 2001. Kyoto, Japan.

37. F. Kon, R.c., M. Roman. *Design and Implementation of Runtime Reflection in Communcation Meddleware: the DynamicTAO Case.* in *In proceedings of ICDCS'99 Workshop on Middleware.* 1999.

38. K. Nahrstedt, H.C., S. Narayan, *QoS-aware Resource Management for Distributed Multimedia Applications.* Journal of High-Speed Networking, Special Issue on Multimedia Networking, 1998. **7**: p. 227-255.

39. N. Parlavantzas, G.C., G. Blair. *A Resource Adaption Framework for Reflective Middleware.* in *Proceedings 2nd International Workshop on Reflective and Adaptive Middleware.* 2003. Rio de Janeiro, Brazil.

40. *OpenORB - A Marriage of three technologies.* 2008 [cited 2008 22/12/2008]; The OpenORB project proposes a philosophy for the development of reflective middleware platforms.]. Available from: http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/meta.php.

41. J. Dowling, T.S., V. Cahill, P. Haraszi, B. Redmond. *Using Reflection to Support Dynamic Adaption of System Software: A Case Study Driven Evaluation.* in *Proceedings of Software Engineering and Reflection 2000.* 2000.

42. P Lee, M.L. *Optimizing ML with Run-Time Code Generation.* in *Proceedings of the 1996 ACM Conference on Programming Language Design and Implementation.* May 1996. Philadelphia, Pennsylvania, USA.

43. L Hornof, T.J. *Certifying Compilation and Run-time Code Generation.* in *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* 1999.

44. V Bala, E.D., S Banerjia. *Dynamo: A Transparent Dynamic Optimization System.* in *Proceedings of the 2000 ACM Conference on Programming Language Design and Implementation.* June 2000. Vancouver, British Columbia, Canada.

45. S Debray, W.E. *Profile-Guided Code Compression.* in *Proceedings of the ACM 2002 Conference on Programming language design and implementation* June 2002. Berlin, Germany.

46. M Madou, B., Anckaert, P Moseley, S Dabray, B De Sutter, K De Bosschere. *Software Protection through Dynamic Code Mutation.* in *Proceedings of the 6th International Workshop on Information Security Applications.* August 2005. Jeju Island, Korea: Springer.

47. C Tschudin, L.Y. *Harnessing Self-Modifying Code for Resilient Software.* in *Proceedings of the 2nd IEEE Workshop on Radical Agent Concepts (WRAC).* September 2005. NASA Goddard Space Flight Center Visitor's Center.

48. Tschudin, C.F. *Fraglets - a Metabolistic Execution Model for Communication Protocols.* in *Proceedings of the 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS).* July 2003. Menlo Park, USA.

49. L Yamamoto, D.S., T Meyer. *Self-Replicating and Self-Modifying Programs in Fraglets.* in *Proceedings of the 2nd International Conference on Bio-Inspired Models of Network, Information, and Computing Systems* December 2007. Budapest, Hungry.

50. H Cai, Z.S., A Vaynberg. *Certified Self-Modifying Code.* in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation.* June 2007. San Diago, California, USA.

51. S S Kulkarni, K.B., *Correctness of Component-based Adaptation,* in *International Symposium on Component-based Software Engineering.* 2004, Springer: Edinburgh, Scotland.

52. J Zhang, B.H.C.C., Z Yang, P K Mckinley *Enabling safe dynamic component-based software adaptation,* in *Architecting Dependable Systems III, Springer Lecture Notes for Computer Science* 2005, Springer. p. 194-211.

53. S Shrivastava, S.W. *Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications.* in *Proceedings of the International Conference on Configurable Distributed Systems* 1998. Washington, DC, USA IEEE.

54. J Magee, J.K., M Sloman, *Constructing distributed systems in Conic.* IEEE Transactions on Software Engineering, 1989. **15**(6): p. 663 - 675.

55. A Tesanovic, M.A., D Nilsson, H Norin, J Hansson, *Ensuring Real-Time Performance Guarantees in Dynamically Reconfigurable Embedded Systems.* Embedded and Ubiquitous Computing, 2005: p. 131-141.

56. M Amirijoo, R.T., T Andersson *Finite horizon QoS prediction of reconfigurable firm real-time systems.* in *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06).* 2006. Las Vegas, Nevada, USA IEEE.

57. P Oreizy, N Medvidovic, R N Taylor, *Architecture-Based Runtime Software Evolution,* in *Proceedings of International Conference on Software Engineering (ICSE).* 1998. p. 117-186.

58. N H Minsky. *Why Should Architectural Principles be Enforced?* in *Proceedings of Computer Security, Dependability and Assurance: From Needs to Solutions.* 1998. York, UK: IEEE.

59. N H Minsky, V Ungureanu, *Law-Governed Interaction: A Coordination and Control Mechanism for Heterogeneous Distributed Systems.* ACM Transactions on Software Engineering and Methodology, 2000. **9**: p. 273-305.

60. M Ionescu, N H Minsky, T D Nguyen. *Enforcement of Communal Policies for P2P Systems.* in *Proceedings of the 6th International Conference on Coordination Models and Languages.* February 2004. Piza Italy.

61. X Ao, N H Minsky. *Flexible Regulation of Distributed Coalitions.* in *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS)*`. October 2003. Norway.

62. N H Minsky, V Ungureanu, W Wang. *Building Reconfiguration Primatives into the Law of the System.* in *Proceedings of the 3rd International Conference on Configurable Distributed Systems.* May 1996. Los Alamitos, California.

63. T Murata, N H Minsky. *On Shouting 'Fire!' - Reglulating Decoupled Communication in Distributed Systems.* in *Proceedings of the International Middleware Conference.* 2003. Rio De Janeiro Brazil: Springer Berlin / Heidelberg.

64. L Wills, S Kannan, S Sander, M Guler, B Heck, J V R Prasad, D Schrage, G Vachtsevanous, *An Open Platform for Reconfiguration Control*, in *IEEE Control Systems Magazine.* June 2001. p. 49-64.

65. J Kramer, J Magee, *The Evolving Philosophers Problem: Dynamic Change Management.* IEEE Transactions on Software Engineering, November 1990. **16**(11): p. 1293-1306.

66. M R Lyu, ed. *Software Fault Tolerance (Trends in Software).* 1995, Wiley Press.

67. *Real-Time Specification for Java 1.0.2.* June 2002 [cited; Available from: http://www.rtsj.org/specjavadoc/book_index.html.

68. D. L. Hall, J.L., *An Introduction to Multisensor Data Fusion.* Proceedings of the IEEE, 1997. **85**: p. 6-23.

69. E. Waltz, J.L., *Multisensor Data Fusion.* 1990: Artech House. 488.

70. N. Bertrand, P.S., *Model Checking Lossy Channels Systems Is Probably Decidable* Foundations of Software Science and Computation Structures, 2003. **2620/2003**: p. 120-135.

71. J D Bradley, M.A.F., P R Miller, P Moxon, A S Wake. *Integrated Modular Avionics – An IMA Design Teams View.* in *IEE Seminar: Certification of Ground / Air System.* 1998.

72. D Fitzjohn, N.T. *Implementing Advanced Avionics* [cited 2003 December 2003]; Available from: http://www.iee.org/OnComms/pn/aerospace/library.cfm.

73. P TH Eugster, P Felber, R Guerraoui, A-M Kermarrec, *The Many Faces of Publish Subscribe.* 2001: Technical Report DSC ID:2000104, EPFL

74. T Bass. *The Federation of Critical Infrastructure Information via Publish-Subscribe Enabled Multisensor Data Fusion.* in *International Conference on Information Fusion.* July 2002: IEEE.

Appendix A

Three Layer VDM Model

This appendix contains the full VDM-SL model described in chapter 2 of this thesis. The model is a three-level model and was developed to provide a basis for exploring the possibilities available within reconfigurable systems. The VDM-SL was particularly aimed to allow IMA type architectures to be expressed, though the model is generic and can express almost any reconfigurable architecture. The operators presented in this model form an extensible reconfiguration language. The model is presented below:

```
--------------------------------
--Model of Aircraft System v4.0--
---Defining possible operators---
--------------------------------


-------------------------------------------------------
--***KEY FOR OPERATION / FUNCTION NAMING CONVENTION***--
-------------------------------------------------------
--
--Processor              -> Procr
--Process                -> Proc
--Activity               -> Act
--Persistent Memory      -> PMem
--Non-Persistent Memory  -> NPMem
--Shared Data            -> SD
--Hardware               -> HW
--Software               -> SW
--With                   -> W
--With Out               -> WO
--Synchronise            -> Sync
--Delete                 -> Del
--Leave Proxies          -> LP
--Location               -> Loc
--
--
--This naming convention is used to shorten function and operation names


-------------
--**STATE**--
-------------


state System of
  HardWare        : Hardware
  SoftWare        : Software
  Loc             : Locations
  SW_HW_Map       : SW_to_HW_Map
  HW_Loc_Map      : HW_to_Loc_Map
inv sys == (forall pid in set dom
             sys.SW_HW_Map.Proc_to_Procrs &
             pid in set dom sys.SoftWare.Processes) and
           (forall cidset in set rng
             sys.SW_HW_Map.Proc_to_Procrs &
             (cidset inter dom sys.HardWare.Cards) = cidset) and
           (forall pid in set dom
```

```
            sys.SW_HW_Map.Proc_to_PMem &
            pid in set dom sys.SoftWare.Processes) and
          (forall cidset in set rng
            sys.SW_HW_Map.Proc_to_PMem &
            (cidset inter dom sys.HardWare.Cards) = cidset) and
          (forall pid in set dom
            sys.SW_HW_Map.Proc_to_NPMem &
            pid in set dom sys.SoftWare.Processes) and
          (forall cidset in set rng
            sys.SW_HW_Map.Proc_to_NPMem &
            (cidset inter dom sys.HardWare.Cards) = cidset) and
          (forall sdid in set dom
            sys.SW_HW_Map.SD_to_NPMem &
            sdid in set dom sys.SoftWare.SDs) and
          (forall cidset in set rng
            sys.SW_HW_Map.SD_to_NPMem &
            (cidset inter dom sys.HardWare.Cards) = cidset) and
          (forall gpid in set dom
             sys.SoftWare.Processes &
             if is_Activity(sys.SoftWare.Processes(gpid)) then
                (sys.SoftWare.Processes(gpid).Loaded = true =>
                (CheckHWConnected(
                   sys.SW_HW_Map.Proc_to_Procrs(gpid),
                   sys.SW_HW_Map.Proc_to_PMem(gpid),
                   sys.SW_HW_Map.Proc_to_NPMem(gpid)) = true) and
                 card (sys.SW_HW_Map.Proc_to_Procrs(gpid)) = 1)
             else
             (sys.SoftWare.Processes(gpid).Activity.Loaded = true =>
                (CheckHWConnected(
                   sys.SW_HW_Map.Proc_to_Procrs(gpid),
                   sys.SW_HW_Map.Proc_to_PMem(gpid),
                   sys.SW_HW_Map.Proc_to_NPMem(gpid)) = true) and
                 card (sys.SW_HW_Map.Proc_to_Procrs(gpid)) = 1))
init sys == sys = mk_System(
                   mk_Hardware({|->}, {|->}, {|->}, {|->}),
                   mk_Software({|->}, {|->}, {|->}, {|->}, {|->}),
                   mk_Locations({|->}),
                   mk_SW_to_HW_Map({|->}, {|->}, {|->}, {|->}),
                   {|->}
                  )
end


-------------
--**TYPES**--
-------------


types


Hardware :: MAUs     : map MAU_ID to MAU
            Cards    : map Card_ID to Card
            Mappings : map Card_ID to MAU_ID
            Linkage  : map HW_Link_ID to HW_Link
inv hw == (forall cid in set dom hw.Mappings &
            cid in set dom hw.Cards) and
          (forall mid in set rng hw.Mappings &
            mid in set dom hw.MAUs) and
          (forall link in set rng hw.Linkage &
            cases link:
                 mk_HW_Uni_Link(a,b) ->
                   a in set dom hw.Cards and b in set dom hw.Cards,
                 mk_HW_Unknown_Link(a,b) ->
                   a in set dom hw.Cards and b in set dom hw.Cards,
                 mk_HW_Bi_Link(mk_HW_Uni_Link(a,b), mk_HW_Uni_Link(aa,bb)) -
>
                   a in set dom hw.Cards and b in set dom hw.Cards and
                   aa in set dom hw.Cards and bb in set dom hw.Cards
```

```
            end);
--Linkage here is only Card linkage (MAU linkage is subsumed in this).
--It could be argued that Linkage is not required when looking at IMA,
--as it assumes a BUS, however failures may require linkage to be evaluated.


Software :: Services       : map Service_ID to set of Global_Process_ID
            Processes      : map Global_Process_ID to Process
            SDs            : map Shared_Data_ID to Shared_Data
            Linkage        : map Link_ID to SW_Link
            SD_Linkage     : map Link_ID to Shared_Data_Link
inv sw == (forall pidset in set rng sw.Services &
            (card(pidset) > 0 and
            ((pidset inter dom sw.Processes) = pidset))) and
          (forall sdlink in set rng sw.SD_Linkage &
            (sdlink.a in set dom sw.Processes and
             sdlink.b in set dom sw.Processes and
             sdlink.Shared_Data in set dom sw.SDs)) and
          (forall link in set rng sw.Linkage &
            cases link:
                 mk_Uni_Link(a,b) ->
                   a in set dom sw.Processes and b in set dom sw.Processes,
                 mk_Unknown_Link(a,b) ->
                   a in set dom sw.Processes and b in set dom sw.Processes,
                 mk_Bi_Link(mk_Uni_Link(a,b), mk_Uni_Link(aa,bb)) ->
                   a in set dom sw.Processes and b in set dom sw.Processes
and
                   aa in set dom sw.Processes and bb in set dom sw.Processes
             end);
--Linkage here is simply Process linkage.  Here the direction of the
--linkage is modelled (Uni, Bi, and Unknown).  Shared Data (SD) Linkage is
--linkage between processes using Shared Data (unidirectional only).


Locations :: Physical_locations : map Loc_ID to Location;
--Locations are modelled very simply within this model.

------------
--Hardware--
------------


--Note: there appears to be the following main levels of dynamic behaviour
--for reconfigurable hardware:
--1) Addind & Removing Cards
--2) Adding & Removing MAUs
--3) Changing Hardware Links
--This model deals with all of the above.

MAU_ID = token;

MAU :: Max_Num_Cards : nat
       Slot_Order    : seq of Interface_Type
inv mau == len mau.Slot_Order = mau.Max_Num_Cards;

Card_ID = token;

Card = Processor | Persistent_Mem | Non_Persistent_Mem;

Processor :: Manufacturer     : token
             Model_Code       : token
             Interface        : Interface_Type
             Speed            : real
             Pipelining       : bool
             Cache            : bool
             Instruction_Set  : Proc_Inst
             Instruction_Ext  : Proc_Inst_Ext
             Co_Processor     : bool
             Register_Size    : Register
```

```
                Max_Op_Temp      : Temperature
                Min_Op_Temp      : Temperature
                Max_Op_Alt       : Altitude
                Max_Humidity     : Humidity;

Persistent_Mem :: Manufacturer      : token
                  Model_Code        : token
                  Interface         : Interface_Type
                  AV_Seek_Time      : MilliSeconds
                  Max_Seek_Time     : MilliSeconds
                  Min_Seek_Time     : MilliSeconds
                  Cache             : bool
                  AV_Transfer_Rate  : MBPerSecond
                  Max_Transfer_Rate : MBPerSecond
                  Min_Transfer_Rate : MBPerSecond
                  Capacity          : Mb
                  Max_Op_Temp       : Temperature
                  Min_Op_Temp       : Temperature
                  Max_Op_Alt        : Altitude
                  Max_Humidity      : Humidity;

Non_Persistent_Mem :: Manufacturer      : token
                      Model_Code        : token
                      Interface         : Interface_Type
                      AV_Transfer_Rate  : MBPerSecond
                      Max_Transfer_Rate : MBPerSecond
                      Min_Transfer_Rate : MBPerSecond
                      Capacity          : Mb
                      Bus_Clock_Rate    : MHz
                      Max_Op_Temp       : Temperature
                      Min_Op_Temp       : Temperature
                      Max_Op_Alt        : Altitude
                      Max_Humidity      : Humidity;

HW_Link = HW_Uni_Link | HW_Bi_Link | HW_Unknown_Link;

HW_Link_ID = token;

HW_Uni_Link :: a : Card_ID
               b : Card_ID
inv uni == uni.a <> uni.b;

HW_Bi_Link :: a : HW_Uni_Link
              b : HW_Uni_Link
inv bi == bi.a.a = bi.b.b and bi.a.b = bi.b.a;

HW_Unknown_Link :: a : Card_ID
                   b : Card_ID
inv uni == uni.a <> uni.b;
--Unknown links are completely unknown in this model.  This means that they
--could be uni-directional, bi-directional, or faulty.

Proc_Inst = <CISC> | <RISC>;

Proc_Inst_Ext = <MMX> | <x3DNow>;
--assuming for simplicity that a processor can only have one special set of
--instructions.

Register = <x32Bit> | <x64Bit>;

Interface_Type = <A> | <B> | <C>;
--can be expanded

Humidity = real;

Temperature = real;

Altitude = real;
```

```
MilliSeconds = real;

MBPerSecond = real;

MHz = real;

Mb = real;


------------
--Software--
------------


--Note: there appears to be 5 main levels of dynamic behaviour for
--reconfigurable software, these are:
--1) Dynamic Linking & Loading + Binary Loading
--2) State and Stack Synchronisation
--3) Interpretation / Execution of intermediately represented code (eg.
Java),
--   this requires a process to execute / interpret the representation
--4) Compilation of code and subsequent execution (can be done either
statically
--   or dynamically)
--5) Changing Software Links
--At present this model copes with 1 through 3 (though 3 is only dealt with
by
--using abstraction to assume that a process can be an intermediate
--representation with its executing / interpreting process).  This section
would
--require further extension to cope with 4.  Also this would need futher
--investigation with respect to resource issues.  The model copes with 5.


Service_ID = token;

Service :: Developer  : token
           Name       : token;

Global_Process_ID = token;

Process = Activity | Proxy | Duplex_Proxy | Condensing_Proxy;

Proxy :: Source   : Global_Process_ID
         Target   : Global_Process_ID
         Activity : Activity
inv proxy == (proxy.Activity.Loaded = false =>
              proxy.Activity.PID = mk_token("null")) and
             (proxy.Activity.Loaded = true =>
              proxy.Activity.PID <> mk_token("null"));

Duplex_Proxy :: Source   : Global_Process_ID
                Target1  : Global_Process_ID
                Target2  : Global_Process_ID
                Activity : Activity
inv proxy == (proxy.Activity.Loaded = false =>
              proxy.Activity.PID = mk_token("null")) and
             (proxy.Activity.Loaded = true =>
              proxy.Activity.PID <> mk_token("null"));

Condensing_Proxy :: Source1 : Global_Process_ID
                    Source2 : Global_Process_ID
                    Target  : Global_Process_ID
                    Activity : Activity
inv proxy == (proxy.Activity.Loaded = false =>
              proxy.Activity.PID = mk_token("null")) and
             (proxy.Activity.Loaded = true =>
              proxy.Activity.PID <> mk_token("null"));
```

```
Activity :: Developer           : token
            Name                : token
            Source              : token
            State               : token
            Initialisation_State : token
            Instruction_Stack   : token
            PID                 : token
            Loaded              : bool
inv act == (act.Loaded = false =>
            act.State = mk_token("null") and
            act.Instruction_Stack = mk_token("null") and
            act.PID = mk_token("null")) and
           (act.Initialisation_State <> mk_token("null")) and
           (act.Loaded = true =>
            act.State <> mk_token("null") and
            act.Instruction_Stack <> mk_token("null") and
            act.PID <> mk_token("null"));
--State represents the variables in the process. Source represents the
source
--binary or intermediate representation of the source code that is executed
/
--interpreted. An intermediate representation would need a process to
interpret
--and execute it (as Java), but this has been abstracted away in this model.
--The Instruction_Stack represents the current execution state.  The
--initialisation state is the state in which the system moves to upon
Loading.
--The PID is the Process ID that the OS / Processor assigns the process.
--The invariant states that it all Processes should have a null State, PID,
and
--Instruction Stack if they are not loaded as it makes no sence for
something to
--have state that does not have memory to store its state.  It also states
--that the oposite is true.
--Activities are single threaded.  Abstractly a set of Activities could
represent
--a multi-threaded process.

SW_Link = Uni_Link | Bi_Link | Unknown_Link;

Link_ID = token;

Uni_Link :: a : Global_Process_ID
            b : Global_Process_ID
inv uni == uni.a <> uni.b;

Bi_Link :: a : Uni_Link
           b : Uni_Link
inv bi == bi.a.a = bi.b.b and bi.a.b = bi.b.a;

Unknown_Link :: a : Global_Process_ID
                b : Global_Process_ID
inv uni == uni.a <> uni.b;
--Unknown links are completely unknown in this model.  They may not even be
--functioning.

Shared_Data_ID = token;

Shared_Data :: Protocol          : Protocol
               State             : token;

Protocol = <Channel> | <Signal> | <Pool> | <Constant> | <Flash_Data> |
  <Overwriting_Buffer> | <Rendezvous> | <Bounded_Buffer> | <Prod> |
  <Stimulus> | <Overwriting_Stim_Buffer> | <Directional_Handshake> |
  <Dataless_Channel> | <Bounded_Stim_Buffer>;

Shared_Data_Link :: a           : Global_Process_ID
                    b           : Global_Process_ID
                    Shared_Data : Shared_Data_ID
```

```
inv uni == uni.a <> uni.b;
--Uni directional from a to b


-------------
--Locations--
-------------


--Note: Locations are not reconfigurable "as such".  Locations are generally
--static.  The following are however possible:
--1) Locations could be added
--2) Locations coule be removed
--3) Locations could change names
--This model can express the above reconfigurations.

Location :: Name     : token
            Coord_x  : real
            Coord_y  : real;

Loc_ID = token;

--The focus of the this model was software reconfiguration, and as such the
--model for Locations has been left very basic.  Locations were modeled to
--highlight the difference between reconfiguration and mobility.  A more
--detailed model of locations could have a tree structure in which a
location
--could be seen as a place in the tree structure, thus Newcastle could be a
--location within England which is also a location etc...  However such a
model
--would have gave no advantage to the purpose of this model.  If a more
complex
--view of locations were to be explored, it would be interesting to capture
--there properties in relation to effects upon the system, for instance the
--temperature which could have an effect on operational requirements for
--systems hardware.


------------------
--Level Mappings--
------------------


--Note: This section of this model represents the mappings between Hardware,
--Software and Location.  Therefore the following reconfigurable behaviour
--is possible:
--1) Processes can move between Hardware (Cards)
--2) Hardware (and thus the Software on it) can move between Locations (this
is
--   more mobility that reconfiguration)
--3) Shared Data can move between Hardware (Cards)
--This model deals with all of the above types of behaviour


SW_to_HW_Map :: Proc_to_Procrs : map Global_Process_ID to set of Card_ID
                Proc_to_PMem   : map Global_Process_ID to set of Card_ID
                Proc_to_NPMem  : map Global_Process_ID to set of Card_ID
                SD_to_NPMem    : map Shared_Data_ID to set of Card_ID
inv swhwmap == (forall pid in set dom swhwmap.Proc_to_Procrs &
                  swhwmap.Proc_to_Procrs(pid) <> {}) and
               (forall pid in set dom swhwmap.Proc_to_PMem &
                  swhwmap.Proc_to_PMem(pid) <> {}) and
               (forall pid in set dom swhwmap.Proc_to_NPMem &
                  swhwmap.Proc_to_NPMem(pid) <> {}) and
               (forall sdid in set dom swhwmap.SD_to_NPMem &
                  swhwmap.SD_to_NPMem(sdid) <> {});

HW_to_Loc_Map = map MAU_ID to Loc_ID;
```

```
--Within the Level Mappings, MAUs are mapped to locations.  Locations could
have
--been mapped to individual cards.  It seemed sensible to do MAUs, as cards
must
--be mapped to MAUs.


-----------------
--**FUNCTIONS**--
-----------------


functions


settoseq : set of Global_Process_ID -> seq of Global_Process_ID
settoseq(s) ==
  cases s:
    {} -> [],
    {x} ->[x],
    s1 union s2 -> (settoseq(s1))^(settoseq(s2))
  end;


settoseqcid : set of Card_ID -> seq of Card_ID
settoseqcid(s) ==
  cases s:
    {} -> [],
    {x} ->[x],
    s1 union s2 -> (settoseq(s1))^(settoseq(s2))
  end;


NumberOfOccur: Interface_Type * seq of Interface_Type -> nat
NumberOfOccur(x,p) == card {i| i in set inds p & p(i) =x};
-- number of occurences of x in p.
-- required to check for correct slots.


CheckProcEquality: Process * Process -> bool
CheckProcEquality(p1, p2) ==
  if is_Activity(p1) then
    p1.Source = p2.Source
  else
    p1.Activity.Source = p2.Activity.Source;


CheckProcLoaded: Process -> bool
CheckProcLoaded(p) ==
  if is_Activity(p) then
    p.Loaded = true
  else
    p.Activity.Loaded = true;


------------------
--**OPERATIONS**--
------------------


operations


-------------------
--Test Operations--
-------------------


TestMakeModel : () ==> ()
```

```
TestMakeModel() ==
  (AddMAU(mauid1, m1);
   AddMAU(mauid2, m2);
   AddCard(cardid1, c1, mauid1);
   AddCard(cardid2, c2, mauid1);
   AddCard(cardid3, c3, mauid2);
   AddCard(cardid4, c4, mauid1);
   AddCard(cardid5, c5, mauid1);
   AddCard(cardid6, c6, mauid1);
   AddCard(cardid7, c7, mauid1);
   AddCard(cardid8, c8, mauid1);
   AddCard(cardid9, c9, mauid1);
   AddCard(cardid10, c10, mauid1);
   AddCard(cardid11, c11, mauid1);
   AddProc(procid1, p1);
   AddProc(procid2, p2);
   AddProc(procid4, p4);
   AddProc(procid5, p5);
   AddSD(sdid1, sd1);
   AddSD(sdid2, sd2);
--   AddSDLink(linkid13, sdid1, procid2, procid1);
--   AddSDLink(linkid20, sdid2, procid1, procid2);
   AssignSDNPMem(cardid8, sdid1);
   AssignSDNPMem(cardid11, sdid2);
   AssignProcProcr(cardid1, procid1);
   AssignProcPMem(cardid4, procid1);
   AssignProcNPMem(cardid6, procid1);
   AssignProcProcr(cardid3, procid4);
   AssignProcPMem(cardid4, procid4);
   AssignProcNPMem(cardid6, procid4);
   AssignProcProcr(cardid2, procid2);
   AssignProcPMem(cardid5, procid2);
   AssignProcNPMem(cardid7, procid2);
   AssignProcProcr(cardid9, procid5);
   AssignProcPMem(cardid5, procid5);
   AssignProcNPMem(cardid7, procid5);
   AddHWBiLink(cardid1, cardid8, linkid1);
   AddHWBiLink(cardid1, cardid4, linkid2);
   AddHWBiLink(cardid1, cardid3, linkid3);
   AddHWBiLink(cardid1, cardid6, linkid4);
   AddHWBiLink(cardid3, cardid4, linkid8);
   AddHWBiLink(cardid3, cardid6, linkid9);
   AddHWBiLink(cardid2, cardid5, linkid10);
   AddHWBiLink(cardid2, cardid7, linkid5);
   AddHWBiLink(cardid3, cardid9, linkid6);
   AddHWBiLink(cardid8, cardid2, linkid7);
   AddHWUniLink(cardid1, cardid3, linkid11);
   AddHWUniLink(cardid3, cardid2, linkid12);
   AddHWBiLink(cardid9, cardid2, linkid14);
   AddHWBiLink(cardid9, cardid5, linkid15);
   AddHWBiLink(cardid9, cardid7, linkid16);
   AddSWUniLink(procid1, procid2, linkid17);
   AddSWUniLink(procid2, procid1, linkid18);
   AddSWBiLink(procid1, procid2, linkid19);

   AddHWBiLink(cardid8, cardid11, linkid21);
   AddHWBiLink(cardid8, cardid10, linkid22);
   AddHWBiLink(cardid10, cardid11, linkid23);

   AddHWBiLink(cardid10, cardid2, linkid24);
   AddHWBiLink(cardid10, cardid5, linkid25);

   LoadProc(procid1);
   LoadProc(procid2);
   LoadProc(procid4);
   LoadProc(procid5);

   AddLoc(location1, locid1);
   AddLoc(location2, locid2);
```

```
    AddLoc(location3, locid3);
    RemoveLoc(locid2);

    ChangeLocName(locid3, mk_token("Gosforth"));
    AssignMAULoc(mauid1, locid1);
    AssignMAULoc(mauid2, locid1);

  );


-----------------------
--Checking Operations--
-----------------------


CheckProcHWConnected : Global_Process_ID ==> bool
CheckProcHWConnected(gpid) ==
  CheckHWConnected(SW_HW_Map.Proc_to_Procrs(gpid),
    SW_HW_Map.Proc_to_PMem(gpid),
    SW_HW_Map.Proc_to_NPMem(gpid));


CheckHWConnected:set of Card_ID * set of Card_ID * set of Card_ID ==> bool
CheckHWConnected(proc, pmem, npmem) ==
  return((CheckAllCardsBiConnected(proc, proc)) and
    (CheckAllCardsBiConnected(proc, pmem)) and
    (CheckAllCardsBiConnected(proc, npmem)));
--This operation is used to check that hardware connections for a given
process
--are valid.  I.e. that the processors are interconnected and that the
--processors have links to the persistent and non-persistent memory.  The
state
--of the system uses this in its invariant.


CheckAllCardsBiConnected: set of Card_ID * set of Card_ID ==> bool
CheckAllCardsBiConnected(cset1, cset2) ==
  return(forall c1 in set cset1 &
    forall c2 in set cset2 &
      c1 <> c2 =>
        exists link in set rng HardWare.Linkage &
          cases link:
            mk_HW_Uni_Link(a,b) -> a = c1 and b = c2 and
              exists link1 in set rng HardWare.Linkage &
                is_HW_Uni_Link(link1) and link1.a = c2 and link1.b = c1,
            mk_HW_Bi_Link(a,-) -> ((a.a = c1 and a.b = c2) or
              (a.a = c2 and a.b = c1)),
            others -> false
          end);
--Others are false as Unknown links may be faulty.


CheckAllCardsUniConnected: set of Card_ID * set of Card_ID ==> bool
CheckAllCardsUniConnected(cset1, cset2) ==
  return(forall c1 in set cset1 &
    forall c2 in set cset2 &
      c1 <> c2 =>
        exists link in set rng HardWare.Linkage &
          cases link:
            mk_HW_Uni_Link(a,b) -> a = c1 and b = c2,
            mk_HW_Bi_Link(a,-) -> ((a.a = c1 and a.b = c2) or
              (a.a = c2 and a.b = c1)),
            others -> false
          end);
--Others are false as Unknown links may be faulty.


FindAllProxies: () ==> set of Global_Process_ID
FindAllProxies() ==
```

```
  return({prox | prox in set dom SoftWare.Processes &
    is_Proxy(SoftWare.Processes(prox)) and
    CheckProcLoaded(SoftWare.Processes(prox))});


FindProxiesFromProc: Global_Process_ID ==> set of Global_Process_ID
FindProxiesFromProc(gpid) ==
  return({prox | prox in set FindAllProxies() &
    if is_Proxy(SoftWare.Processes(gpid)) then
      SoftWare.Processes(prox).Source = gpid and
      SoftWare.Processes(gpid).Target = prox
    else if is_Duplex_Proxy(SoftWare.Processes(gpid)) then
      SoftWare.Processes(prox).Source = gpid and
      (SoftWare.Processes(gpid).Target1 = prox or
      SoftWare.Processes(gpid).Target2 = prox)
    else if is_Condensing_Proxy(SoftWare.Processes(gpid)) then
      SoftWare.Processes(prox).Source = gpid and
      SoftWare.Processes(gpid).Target = prox
    else
      true
  });


FindValidProxies : set of Global_Process_ID * Global_Process_ID *
Global_Process_ID * seq of Global_Process_ID ==> set of Global_Process_ID
FindValidProxies(beenset, current, target, todo) ==
  return({prox | prox in set FindProxiesFromProc(current) &
    prox not in set beenset and
    prox not in set elems todo and
    CheckAllCardsUniConnected(
      SW_HW_Map.Proc_to_Procrs(current),
      SW_HW_Map.Proc_to_Procrs(prox))
  });


FindAllDuplexProxies: () ==> set of Global_Process_ID
FindAllDuplexProxies() ==
  return({prox | prox in set dom SoftWare.Processes &
    is_Duplex_Proxy(SoftWare.Processes(prox)) and
    CheckProcLoaded(SoftWare.Processes(prox))});


FindDuplexProxiesFromProc: Global_Process_ID ==> set of Global_Process_ID
FindDuplexProxiesFromProc(gpid) ==
  return({prox | prox in set FindAllDuplexProxies() &
    if is_Proxy(SoftWare.Processes(gpid)) then
      SoftWare.Processes(prox).Source = gpid and
      SoftWare.Processes(gpid).Target = prox
    else if is_Duplex_Proxy(SoftWare.Processes(gpid)) then
      SoftWare.Processes(prox).Source = gpid and
      (SoftWare.Processes(gpid).Target1 = prox or
      SoftWare.Processes(gpid).Target2 = prox)
    else if is_Condensing_Proxy(SoftWare.Processes(gpid)) then
      SoftWare.Processes(prox).Source = gpid and
      SoftWare.Processes(gpid).Target = prox
    else
      true
  });


FindValidDuplexProxies : set of Global_Process_ID * Global_Process_ID *
Global_Process_ID * seq of Global_Process_ID ==> set of Global_Process_ID
FindValidDuplexProxies(beenset, current, target, todo) ==
  return({prox | prox in set FindDuplexProxiesFromProc(current) &
    prox not in set beenset and
    prox not in set elems todo and
    CheckAllCardsUniConnected(
      SW_HW_Map.Proc_to_Procrs(current),
      SW_HW_Map.Proc_to_Procrs(prox))
```

```
  });


FindAllCondensingProxies: () ==> set of Global_Process_ID
FindAllCondensingProxies() ==
  return({prox | prox in set dom SoftWare.Processes &
    is_Condensing_Proxy(SoftWare.Processes(prox)) and
    CheckProcLoaded(SoftWare.Processes(prox))});


FindCondensingProxiesFromProc: Global_Process_ID ==> set of
Global_Process_ID
FindCondensingProxiesFromProc(gpid) ==
  return({prox | prox in set FindAllCondensingProxies() &
    if is_Proxy(SoftWare.Processes(gpid)) then
      (SoftWare.Processes(prox).Source1 = gpid or
      SoftWare.Processes(prox).Source2 = gpid) and
      SoftWare.Processes(gpid).Target = prox
    else if is_Duplex_Proxy(SoftWare.Processes(gpid)) then
      (SoftWare.Processes(prox).Source1 = gpid or
      SoftWare.Processes(prox).Source2 = gpid) and
      (SoftWare.Processes(gpid).Target1 = prox or
      SoftWare.Processes(gpid).Target2 = prox)
    else if is_Condensing_Proxy(SoftWare.Processes(gpid)) then
      (SoftWare.Processes(prox).Source1 = gpid or
      SoftWare.Processes(prox).Source2 = gpid) and
      SoftWare.Processes(gpid).Target = prox
    else
      true
  });


FindValidCondensingProxies : set of Global_Process_ID * Global_Process_ID *
Global_Process_ID * seq of Global_Process_ID ==> set of Global_Process_ID
FindValidCondensingProxies(beenset, current, target, todo) ==
  return({prox | prox in set FindCondensingProxiesFromProc(current) &
    prox not in set beenset and
    prox not in set elems todo and
    CheckAllCardsUniConnected(
      SW_HW_Map.Proc_to_Procrs(current),
      SW_HW_Map.Proc_to_Procrs(prox))
  });


FindValidActivities : set of Global_Process_ID * Global_Process_ID *
Global_Process_ID * seq of Global_Process_ID ==> set of Global_Process_ID
FindValidActivities(beenset, current, target, todo) ==
  if (is_Proxy(SoftWare.Processes(current)) or
    is_Condensing_Proxy(SoftWare.Processes(current))) and
    SoftWare.Processes(current).Target = target and
    CheckAllCardsUniConnected(
      SW_HW_Map.Proc_to_Procrs(current),
      SW_HW_Map.Proc_to_Procrs(target)) then
    return {target}
  else if is_Duplex_Proxy(SoftWare.Processes(current)) and
    (SoftWare.Processes(current).Target1 = target or
      SoftWare.Processes(current).Target2 = target) and
    CheckAllCardsUniConnected(
      SW_HW_Map.Proc_to_Procrs(current),
      SW_HW_Map.Proc_to_Procrs(target)) then
    return {target}
  else
    return{};


FindSDsFromProc: Global_Process_ID ==> set of Link_ID
FindSDsFromProc(gpid) ==
  return({lid | lid in set dom SoftWare.SD_Linkage &
    SoftWare.SD_Linkage(lid).a = gpid});
```

```
FindValidSDs : set of Global_Process_ID * Global_Process_ID *
Global_Process_ID * seq of Global_Process_ID ==> set of Global_Process_ID
FindValidSDs(beenset, current, target, todo) ==
  return({SoftWare.SD_Linkage(lid).b | lid in set FindSDsFromProc(current) &
    SoftWare.SD_Linkage(lid).b not in set beenset and
    SoftWare.SD_Linkage(lid).b not in set elems todo and
    SoftWare.SD_Linkage(lid).b = target and
    CheckAllCardsUniConnected(
      SW_HW_Map.Proc_to_Procrs(SoftWare.SD_Linkage(lid).a),
      SW_HW_Map.SD_to_NPMem(SoftWare.SD_Linkage(lid).Shared_Data)) and
    CheckAllCardsUniConnected(
      SW_HW_Map.SD_to_NPMem(SoftWare.SD_Linkage(lid).Shared_Data),
      SW_HW_Map.Proc_to_Procrs(SoftWare.SD_Linkage(lid).b))
    });


CheckIsRoute: Global_Process_ID * Global_Process_ID ==> bool
CheckIsRoute(gpidA, gpidB) ==
  return(CheckAllCardsUniConnected(
    SW_HW_Map.Proc_to_Procrs(gpidA),
    SW_HW_Map.Proc_to_Procrs(gpidB)) or
  CheckProxyRoute(gpidB, {}, [gpidA]))
pre gpidA <> gpidB;


CheckProxyRoute : Global_Process_ID * set of Global_Process_ID * seq of
Global_Process_ID ==> bool
CheckProxyRoute(target, beenset, togoseq) ==
  if togoseq = [] then return false
  else
    if (exists x in set (
      FindValidProxies(beenset, hd togoseq, target, tl togoseq) union
      FindValidSDs(beenset, hd togoseq, target, tl togoseq) union
      FindValidActivities(beenset, hd togoseq, target, tl togoseq) union
      FindValidDuplexProxies(beenset, hd togoseq, target, tl togoseq) union
      FindValidCondensingProxies(beenset, hd togoseq, target, tl togoseq)) &
      x = target) then return true
    else
      return CheckProxyRoute(target, {hd togoseq} union beenset,
        tl togoseq ^ settoseq(
        FindValidProxies(beenset, hd togoseq, target, tl togoseq) union
        FindValidSDs(beenset, hd togoseq, target, tl togoseq) union
        FindValidActivities(beenset, hd togoseq, target, tl togoseq) union
        FindValidDuplexProxies(beenset, hd togoseq, target, tl togoseq)
union
        FindValidCondensingProxies(beenset, hd togoseq, target, tl
togoseq)));


CheckProcsHaveConnection: Global_Process_ID * Global_Process_ID ==> bool
CheckProcsHaveConnection(A, B) ==
  return(
    exists link in set rng SoftWare.Linkage &
      cases link:
        mk_Uni_Link(a,b) -> a = A and b = B and
          exists link1 in set rng SoftWare.Linkage &
            is_Uni_Link(link1) and link1.a = B and link1.b = A,
        mk_Bi_Link(a,-) -> ((a.a = A and a.b = B) or
          (a.a = B and a.b = A)),
        others -> false
      end and
    CheckIsRoute(A,B) and
    CheckIsRoute(B,A)
  );
--Others are false as Unknown links may be faulty.
--this operation checks that a state copy and synch can be physically done.
```

```
--It checks that there exists hardware connections between the Processors
for
--the Processes.


CheckSWConnections: Global_Process_ID ==> bool
CheckSWConnections(gpid) ==
  return(
    forall link in set rng SoftWare.Linkage &
      cases link:
        mk_Uni_Link(a,b) -> a = gpid or b = gpid =>
          CheckIsRoute(a,b),
        mk_Bi_Link(a,-) -> a.a = gpid or a.b = gpid =>
          CheckIsRoute(a.a,a.b) and CheckIsRoute(a.b,a.a)
      end);


CheckProcrsConnect: Global_Process_ID * seq of Card_ID ==> bool
CheckProcrsConnect(gpid, cids) ==
  return (forall p1 in set SW_HW_Map.Proc_to_Procrs(gpid) &
    forall p2 in set (inds cids) &
      (p1 <> cids(p2) and is_Processor(HardWare.Cards(cids(p2)))) =>
        exists link in set rng HardWare.Linkage &
          cases link:
            mk_HW_Uni_Link(a,b) -> a = p1 and b = cids(p2) and
              exists link1 in set rng HardWare.Linkage &
                is_HW_Uni_Link(link1) and link1.a = cids(p2) and link1.b =
p1,
            mk_HW_Bi_Link(a,-) -> ((a.a = p1 and a.b = cids(p2)) or
              (a.a = cids(p2) and a.b = p1)),
            others -> false
          end);
--this operation checks that a move or copy can be physically done.  It
checks
--that there exists hardware connections between the existing processors and
the
--new processors which the process will be assigned to.


CheckRightCardTypes : seq of Card_ID ==> bool
CheckRightCardTypes(cids) ==
  return((forall p in set(inds cids) &
    cids(p) in set dom HardWare.Cards) and
  (card{cids(p)|p in set(inds cids) &
    is_Processor(HardWare.Cards(cids(p)))} > 0) and
  (card{cids(p)|p in set(inds cids) &
    is_Persistent_Mem(HardWare.Cards(cids(p)))} > 0) and
  (card{cids(p)|p in set(inds cids) &
    is_Non_Persistent_Mem(HardWare.Cards(cids(p)))} > 0));
--this operation checks that given a sequence of cards, it has the required
--cards to execute a process (i.e. a processor, persistent memory and
--non-persistent memory.


CheckCopySDState : Shared_Data_ID * Shared_Data_ID ==> bool
CheckCopySDState(A, B) ==
  return(forall m1 in set SW_HW_Map.SD_to_NPMem(A) &
    forall m2 in set SW_HW_Map.SD_to_NPMem(B) &
      (m1 <> m2) =>
        exists linka in set rng HardWare.Linkage &
          cases linka:
            mk_HW_Uni_Link(a,b) -> (a = m1 and
              is_Processor(HardWare.Cards(b))) and
                exists link1a in set rng HardWare.Linkage &
                  (is_HW_Uni_Link(link1a) and link1a.a = linka.b
                    and link1a.b = linka.a) and
                    exists linkb in set rng HardWare.Linkage &
                      cases linkb:
                        mk_HW_Uni_Link(aa,bb) -> aa = m2 and bb = b and
```

```
                        exists link1b in set rng HardWare.Linkage &
                          (is_HW_Uni_Link(link1b) and link1b.a = linkb.b
                            and link1b.b = linkb.a),
                      mk_HW_Bi_Link(aa,-) -> ((aa.a = m2 and aa.b = b) or
                        (aa.a = b and aa.b = m2)),
                      others -> false
                    end,
          mk_HW_Bi_Link(a,-) -> ((a.a = m1 and
            is_Processor(HardWare.Cards(a.b))) or
              (is_Processor(HardWare.Cards(a.a)) and a.b = m1)) and
                 exists linkb in set rng HardWare.Linkage &
                    cases linkb:
                      mk_HW_Uni_Link(aa,bb) -> aa = m2 and (
                        bb = a.a or bb = a.b) and
                          exists link1b in set rng HardWare.Linkage &
                            is_HW_Uni_Link(link1b) and link1b.a = linkb.b
                              and link1b.b = linkb.a,
                      mk_HW_Bi_Link(aa,-) ->
                        (aa.a = m2 or aa.b = m2) and
                          ((aa.b = a.a and
                            is_Processor(HardWare.Cards(aa.b))) or
                           (aa.a = a.a and
                            is_Processor(HardWare.Cards(aa.a))) or
                           (aa.b = a.b and
                            is_Processor(HardWare.Cards(aa.b))) or
                           (aa.a = a.b and
                            is_Processor(HardWare.Cards(aa.a)))),
                      others -> false
                  end,
            others -> false
          end);
--This ensures that there are connections between the Non-Persistent Memory
used for
--each SD, via a processor.


---------------------------------
--Model Construction Operations--
---------------------------------


AddMAU : MAU_ID * MAU ==> ()
AddMAU (mauid, mau) ==
  HardWare.MAUs := HardWare.MAUs munion {mauid |-> mau}
  pre mauid not in set dom HardWare.MAUs;


RemoveMAU : MAU_ID ==> ()
RemoveMAU (mauid) ==
  HardWare.MAUs := {mauid} <-: HardWare.MAUs
  pre mauid not in set rng HardWare.Mappings and
      mauid in set dom HardWare.MAUs;
--The above pre condition only checks for no cards in the MAU and thus no
--executing software, but may want to allow cards to be in while removed,
--but only check for no running software.  If an MAU is removed with cards
--in this model says nothing.


AddCard : Card_ID * Card * MAU_ID ==> ()
AddCard (cid1, card1, mid1) ==
  HardWare :=
    mk_Hardware(HardWare.MAUs,
      HardWare.Cards munion {cid1 |-> card1},
      HardWare.Mappings munion {cid1 |-> mid1},
      HardWare.Linkage)
  pre cid1 not in set dom HardWare.Cards and
      mid1 in set dom HardWare.MAUs and
      (NumberOfOccur(card1.Interface, HardWare.MAUs(mid1).Slot_Order) >
      (card {i|i in set dom HardWare.Cards & HardWare.Mappings(i) = mid1
```

```
      and HardWare.Cards(i).Interface = card1.Interface}));
--mid1 in set dom HardWare.MAUs only in pre condition to stop run time
errors,
--as invarient over hardware will do the same job


RemoveCard : Card_ID ==> ()
RemoveCard (cid) ==
  HardWare :=
    mk_Hardware(HardWare.MAUs,
      {cid} <-: HardWare.Cards,
      {cid} <-: HardWare.Mappings,
      HardWare.Linkage)
  pre cid in set dom HardWare.Cards and
      cid not in set dunion rng SW_HW_Map.Proc_to_Procrs and
      cid not in set dunion rng SW_HW_Map.Proc_to_PMem and
      cid not in set dunion rng SW_HW_Map.Proc_to_NPMem and
      forall link in set rng HardWare.Linkage &
        cases link:
          mk_HW_Uni_Link(a,b) -> cid <> a and cid <> b,
          mk_HW_Unknown_Link(a,b) -> cid <> a and cid <> b,
          mk_HW_Bi_Link(mk_HW_Uni_Link(a,b), mk_HW_Uni_Link(aa,bb)) ->
            cid <> a and cid <> b and cid <> aa and cid <> bb
        end;
--pre condition checks for no software running on card before removing.  It
--also checks that the card has no linkage before removing it.
--If a card is removed while having software on it this model says nothing
--about the outcome.  Also if a card is removed while it has linkage, the
--model says nothing about it.


AddHWUniLink : Card_ID * Card_ID * HW_Link_ID ==> ()
AddHWUniLink(cid1, cid2, lid) ==
  HardWare.Linkage := HardWare.Linkage munion
    {lid |-> mk_HW_Uni_Link(cid1, cid2)}
  pre lid not in set dom HardWare.Linkage;


AddHWUnknownLink : Card_ID * Card_ID * HW_Link_ID ==> ()
AddHWUnknownLink(cid1, cid2, lid) ==
  HardWare.Linkage := HardWare.Linkage munion
    {lid |-> mk_HW_Unknown_Link(cid1, cid2)}
  pre lid not in set dom HardWare.Linkage;


AddHWBiLink : Card_ID * Card_ID * HW_Link_ID ==> ()
AddHWBiLink(cid1, cid2, lid) ==
  HardWare.Linkage := HardWare.Linkage munion
    {lid |-> mk_HW_Bi_Link(mk_HW_Uni_Link(cid1, cid2),
    mk_HW_Uni_Link(cid2, cid1))}
  pre lid not in set dom HardWare.Linkage;


RemoveHWLink : HW_Link_ID ==> ()
RemoveHWLink(lid) ==
  HardWare.Linkage := {lid} <-: HardWare.Linkage
  pre lid in set dom HardWare.Linkage;


AddProc : Global_Process_ID * Process ==> ()
AddProc(gpid, proc) ==
  SoftWare.Processes := SoftWare.Processes munion {gpid |-> proc}
  pre gpid not in set dom SoftWare.Processes;


RemoveProc : Global_Process_ID ==> ()
RemoveProc(gpid) ==
  SoftWare.Processes := {gpid} <-: SoftWare.Processes
  pre gpid in set dom SoftWare.Processes and
```

```
    forall procsets in set rng SoftWare.Services &
      gpid not in set procsets and
    forall SDlink in set rng SoftWare.SD_Linkage &
      SDlink.a <> gpid and SDlink.b <> gpid and
    forall link in set rng SoftWare.Linkage &
      cases link:
        mk_Uni_Link(a,b) -> gpid <> a and gpid <> b,
        mk_Unknown_Link(a,b) -> gpid <> a and gpid <> b,
        mk_Bi_Link(mk_Uni_Link(a,b), mk_Uni_Link(aa,bb)) ->
          gpid <> a and gpid <> b and gpid <> aa and gpid <> bb
      end;
--pre condition checks that the process has no linkage before removing it.
--If a process is removed while it has linkage, the model says nothing about
it.
--pre condition also checks that the process is not the member of a service.


AddService : Service_ID * set of Global_Process_ID ==> ()
AddService(serv_id, serv) ==
  SoftWare.Services := SoftWare.Services munion {serv_id |-> serv}
  pre serv_id not in set dom SoftWare.Services;


AddProcToService : Service_ID * Global_Process_ID ==> ()
AddProcToService(serv_id, gpid) ==
  SoftWare.Services := SoftWare.Services ++
    {serv_id |-> SoftWare.Services(serv_id) union {gpid}}
  pre serv_id in set dom SoftWare.Services and
      gpid not in set SoftWare.Services(serv_id);


RemoveProcFromService : Service_ID * Global_Process_ID ==> ()
RemoveProcFromService(serv_id, gpid) ==
  if card(SoftWare.Services(serv_id)) = 1
  then
    SoftWare.Services := {serv_id} <-: SoftWare.Services
  else
    SoftWare.Services := SoftWare.Services ++
      {serv_id |-> SoftWare.Services(serv_id) \ {gpid}}
  pre serv_id in set dom SoftWare.Services and
      gpid in set SoftWare.Services(serv_id);
--no remove service operator as when the last process is removed, the
service
--is no more.


AddSD : Shared_Data_ID * Shared_Data ==> ()
AddSD(id, ida) ==
  SoftWare.SDs := SoftWare.SDs munion {id |-> ida}
  pre id not in set dom SoftWare.SDs;


RemoveSD : Shared_Data_ID ==> ()
RemoveSD(id) ==
  SoftWare.SDs := {id} <-: SoftWare.SDs
  pre id in set dom SoftWare.SDs and
      forall SDlink in set rng SoftWare.SD_Linkage &
        SDlink.Shared_Data <> id;
--pre condition checks that the Shared Data has no linkage before removing
it.
--If a Shared Data is removed while it has linkage, the model says nothing
--about it.


AddSDLink: Link_ID * Shared_Data_ID * Global_Process_ID *
Global_Process_ID==>()
AddSDLink(lid, sdid, gpid1, gpid2) ==
  SoftWare.SD_Linkage :=
```

```
    SoftWare.SD_Linkage munion {lid |-> mk_Shared_Data_Link(gpid1, gpid2,
sdid)}
  pre lid not in set dom SoftWare.SD_Linkage and
      lid not in set dom SoftWare.Linkage;
--This does not check for underlying network support for the link, as it is
--assumed that software links can be added that are not supported by the
--hardware.


RemoveSDLink : Link_ID ==> ()
RemoveSDLink(lid) ==
  SoftWare.SD_Linkage := {lid} <-: SoftWare.SD_Linkage
  pre lid in set dom SoftWare.SD_Linkage;


AddSWUniLink : Global_Process_ID * Global_Process_ID * Link_ID ==> ()
AddSWUniLink(gpid1, gpid2, lid) ==
  SoftWare.Linkage :=
    SoftWare.Linkage munion {lid |-> mk_Uni_Link(gpid1, gpid2)}
  pre lid not in set dom SoftWare.SD_Linkage and
      lid not in set dom SoftWare.Linkage;
--This does not check for underlying network support for the link, as it is
--assumed that software links can be added that are not supported by the
--hardware.


AddSWUnknownLink : Global_Process_ID * Global_Process_ID * Link_ID ==> ()
AddSWUnknownLink(gpid1, gpid2, lid) ==
  SoftWare.Linkage :=
SoftWare.Linkage munion {lid |-> mk_Unknown_Link(gpid1, gpid2)}
  pre lid not in set dom SoftWare.SD_Linkage and
      lid not in set dom SoftWare.Linkage;
--This does not check for underlying network support for the link, as it is
--assumed that software links can be added that are not supported by the
--hardware.


AddSWBiLink : Global_Process_ID * Global_Process_ID * Link_ID ==> ()
AddSWBiLink(gpid1, gpid2, lid) ==
  SoftWare.Linkage :=
    SoftWare.Linkage munion {lid |-> mk_Bi_Link(mk_Uni_Link(gpid1, gpid2),
      mk_Uni_Link(gpid2, gpid1))}
  pre lid not in set dom SoftWare.SD_Linkage and
      lid not in set dom SoftWare.Linkage;
--This does not check for underlying network support for the link, as it is
--assumed that software links can be added that are not supported by the
--hardware.


RemoveSWLink : Link_ID ==> ()
RemoveSWLink(lid) ==
  SoftWare.Linkage := {lid} <-: SoftWare.Linkage
  pre lid in set dom SoftWare.Linkage;


RePointSWLinks : Global_Process_ID * Global_Process_ID ==> ()
RePointSWLinks (oldid, newid) ==
  for all x in set dom SoftWare.Linkage do
    (if (is_Uni_Link(SoftWare.Linkage(x)) or
is_Unknown_Link(SoftWare.Linkage(x)))
      and SoftWare.Linkage(x).a = oldid then
        SoftWare.Linkage := SoftWare.Linkage ++
          {x |-> mk_Uni_Link(newid, SoftWare.Linkage(x).b)}
    else if (is_Uni_Link(SoftWare.Linkage(x)) or
is_Unknown_Link(SoftWare.Linkage(x)))
      and SoftWare.Linkage(x).b = oldid then
        SoftWare.Linkage := SoftWare.Linkage ++
          {x |-> mk_Uni_Link(SoftWare.Linkage(x).a, newid)}
```

```
    else if is_Bi_Link(SoftWare.Linkage(x)) and SoftWare.Linkage(x).a.a =
oldid then
      SoftWare.Linkage := SoftWare.Linkage ++
        {x |-> mk_Bi_Link(mk_Uni_Link(newid, SoftWare.Linkage(x).a.b),
          mk_Uni_Link(SoftWare.Linkage(x).b.a, newid))}
    else if is_Bi_Link(SoftWare.Linkage(x)) and SoftWare.Linkage(x).a.b =
oldid then
      SoftWare.Linkage := SoftWare.Linkage ++
        {x |-> mk_Bi_Link(mk_Uni_Link(SoftWare.Linkage(x).a.a, newid),
          mk_Uni_Link(newid, SoftWare.Linkage(x).b.b))}
    );

RePointSDLinks : Global_Process_ID * Global_Process_ID ==> ()
RePointSDLinks (oldid, newid) ==
   for all x in set dom SoftWare.SD_Linkage do
    (if SoftWare.SD_Linkage(x).a = oldid then
      SoftWare.SD_Linkage := SoftWare.SD_Linkage ++
        {x |-> mk_Shared_Data_Link(newid, SoftWare.SD_Linkage(x).b,
          SoftWare.SD_Linkage(x).Shared_Data)}
    else if SoftWare.SD_Linkage(x).b = oldid then
      SoftWare.SD_Linkage := SoftWare.SD_Linkage ++
        {x |-> mk_Shared_Data_Link(SoftWare.SD_Linkage(x).a, newid,
          SoftWare.SD_Linkage(x).Shared_Data)}
    );


AssignProcProcr : Card_ID * Global_Process_ID ==> ()
AssignProcProcr(CID, GPID) ==
  if GPID in set dom SW_HW_Map.Proc_to_Procrs
  then
    SW_HW_Map.Proc_to_Procrs :=
      SW_HW_Map.Proc_to_Procrs ++
        {GPID |-> SW_HW_Map.Proc_to_Procrs(GPID) union {CID}}
  else
    SW_HW_Map.Proc_to_Procrs :=
      SW_HW_Map.Proc_to_Procrs munion {GPID |-> {CID}}
  pre CID in set dom HardWare.Cards and
      is_Processor(HardWare.Cards(CID));
--CID in set dom HardWare.Cards only in pre condition to stop run time
errors,
--as invariant over hardware will do the same job


AssignProcPMem : Card_ID * Global_Process_ID ==> ()
AssignProcPMem(CID, GPID) ==
  if GPID in set dom SW_HW_Map.Proc_to_PMem
  then
    SW_HW_Map.Proc_to_PMem :=
      SW_HW_Map.Proc_to_PMem ++
        {GPID |-> SW_HW_Map.Proc_to_PMem(GPID) union {CID}}
  else
    SW_HW_Map.Proc_to_PMem :=
      SW_HW_Map.Proc_to_PMem munion {GPID |-> {CID}}
  pre CID in set dom HardWare.Cards and
      is_Persistent_Mem(HardWare.Cards(CID));
--CID in set dom HardWare.Cards only in pre condition to stop run time
errors,
--as invariant over hardware will do the same job


AssignProcNPMem : Card_ID * Global_Process_ID ==> ()
AssignProcNPMem(CID, GPID) ==
  if GPID in set dom SW_HW_Map.Proc_to_NPMem
  then
    SW_HW_Map.Proc_to_NPMem :=
      SW_HW_Map.Proc_to_NPMem ++
        {GPID |-> SW_HW_Map.Proc_to_NPMem(GPID) union {CID}}
  else
    SW_HW_Map.Proc_to_NPMem :=
```

```
      SW_HW_Map.Proc_to_NPMem munion {GPID |-> {CID}}
  pre CID in set dom HardWare.Cards and
      is_Non_Persistent_Mem(HardWare.Cards(CID));
--CID in set dom HardWare.Cards only in pre condition to stop run time
errors,
--as invariant over hardware will do the same job


UnAssignProcCard : Global_Process_ID ==> ()
UnAssignProcCard(GPID) ==
  SW_HW_Map :=
    mk_SW_to_HW_Map(
      {GPID} <-: SW_HW_Map.Proc_to_Procrs,
      {GPID} <-: SW_HW_Map.Proc_to_PMem,
      {GPID} <-: SW_HW_Map.Proc_to_NPMem,
      SW_HW_Map.SD_to_NPMem)
  pre GPID in set dom SoftWare.Processes and
      SoftWare.Processes(GPID).Loaded = false and
      (GPID in set dom SW_HW_Map.Proc_to_Procrs or
      GPID in set dom SW_HW_Map.Proc_to_PMem or
      GPID in set dom SW_HW_Map.Proc_to_NPMem);
--pre condition stops the un-assignment of a processes cards while the
processes
--are loaded upon them.


AssignSDNPMem : Card_ID * Shared_Data_ID ==> ()
AssignSDNPMem(CID, SDID) ==
  if SDID in set dom SW_HW_Map.SD_to_NPMem
  then
    SW_HW_Map.SD_to_NPMem :=
      SW_HW_Map.SD_to_NPMem ++
        {SDID |-> SW_HW_Map.SD_to_NPMem(SDID) union {CID}}
  else
    SW_HW_Map.SD_to_NPMem :=
      SW_HW_Map.SD_to_NPMem munion {SDID |-> {CID}}
  pre CID in set dom HardWare.Cards and
      is_Non_Persistent_Mem(HardWare.Cards(CID));
--CID in set dom HardWare.Cards only in pre condition to stop run time
errors,
--as invariant over hardware will do the same job


UnAssignSDCard : Shared_Data_ID ==> ()
UnAssignSDCard(SDID) ==
  SW_HW_Map.SD_to_NPMem :=
    {SDID} <-: SW_HW_Map.SD_to_NPMem
  pre SDID in set dom SW_HW_Map.SD_to_NPMem;


AssignProcCards : Global_Process_ID * seq of Card_ID ==> ()
AssignProcCards(gpid, cids) ==
  for i = 1 to card(inds cids) do
    if is_Processor(HardWare.Cards(cids(i)))
    then
      AssignProcProcr(cids(i), gpid)
    else
      if is_Persistent_Mem(HardWare.Cards(cids(i)))
      then
        AssignProcPMem(cids(i), gpid)
      else
        AssignProcNPMem(cids(i), gpid)
  pre gpid in set dom SoftWare.Processes;
--this allows a sequence of cards to be assigned to a process


ChangeProxyTarget: Global_Process_ID * Global_Process_ID ==> ()
ChangeProxyTarget(source, newtarget) ==
  SoftWare.Processes(source).Target := newtarget;
```

```
ChangeCondensingProxyTarget: Global_Process_ID * Global_Process_ID ==> ()
ChangeCondensingProxyTarget(source, newtarget) ==
  SoftWare.Processes(source).Target := newtarget;


ChangeDuplexProxyTarget: Global_Process_ID * Global_Process_ID *
Global_Process_ID ==> ()
ChangeDuplexProxyTarget(source, oldid, newid) ==
  if SoftWare.Processes(source).Target1 = oldid then
    SoftWare.Processes(source).Target1 := newid
  else
    SoftWare.Processes(source).Target2 := newid
pre oldid = SoftWare.Processes(source).Target1 or oldid =
SoftWare.Processes(source).Target2;


ChangeProxySource: Global_Process_ID * Global_Process_ID ==> ()
ChangeProxySource(proxyid, newsource) ==
  SoftWare.Processes(proxyid).Source := newsource;


ChangeDuplexProxySource: Global_Process_ID * Global_Process_ID ==> ()
ChangeDuplexProxySource(proxyid, newsource) ==
  SoftWare.Processes(proxyid).Source := newsource;


ChangeCondensingProxySource: Global_Process_ID * Global_Process_ID *
Global_Process_ID ==> ()
ChangeCondensingProxySource(proxyid, oldid, newid) ==
  if SoftWare.Processes(proxyid).Source1 = oldid then
    SoftWare.Processes(proxyid).Source1 := newid
  else
    SoftWare.Processes(proxyid).Source2 := newid
pre oldid = SoftWare.Processes(proxyid).Source1 or oldid =
SoftWare.Processes(proxyid).Source2;


AddSameSWLinksAsProc : Global_Process_ID * Global_Process_ID ==> ()
AddSameSWLinksAsProc (oldid, newid) ==
  for all x in set dom SoftWare.Linkage do
    (if (is_Uni_Link(SoftWare.Linkage(x)) or
is_Unknown_Link(SoftWare.Linkage(x)))
      and SoftWare.Linkage(x).a = oldid then
      SoftWare.Linkage := SoftWare.Linkage munion
      {mk_token([oldid, newid, card(dom SoftWare.Linkage)])
        |-> mk_Uni_Link(newid, SoftWare.Linkage(x).b)}
    else if (is_Uni_Link(SoftWare.Linkage(x)) or
      is_Unknown_Link(SoftWare.Linkage(x)))
      and SoftWare.Linkage(x).b = oldid then
      SoftWare.Linkage := SoftWare.Linkage munion
      {mk_token([oldid, newid, card(dom SoftWare.Linkage)])
        |-> mk_Uni_Link(SoftWare.Linkage(x).a, newid)}
    else if is_Bi_Link(SoftWare.Linkage(x)) and
      SoftWare.Linkage(x).a.a = oldid then
      SoftWare.Linkage := SoftWare.Linkage munion
      {mk_token([oldid, newid, card(dom SoftWare.Linkage)])
        |-> mk_Bi_Link(mk_Uni_Link(newid, SoftWare.Linkage(x).a.b),
        mk_Uni_Link(SoftWare.Linkage(x).b.a, newid))}
    else if is_Bi_Link(SoftWare.Linkage(x)) and SoftWare.Linkage(x).a.b =
oldid then
      SoftWare.Linkage := SoftWare.Linkage munion
      {mk_token([oldid, newid, card(dom SoftWare.Linkage)])
        |-> mk_Bi_Link(mk_Uni_Link(SoftWare.Linkage(x).a.a, newid),
        mk_Uni_Link(newid, SoftWare.Linkage(x).b.b))}
    );
--the generated linkids are not perfect, but will serfice for this model
--(a better method should be used in an implementation).
```

```
--Have not done SD Links here.


UnloadedProc : Global_Process_ID ==> Process
UnloadedProc(gpid) ==
  if is_Activity(SoftWare.Processes(gpid))
  then
  return(
    mk_Activity(SoftWare.Processes(gpid).Developer,
      SoftWare.Processes(gpid).Name,
      SoftWare.Processes(gpid).Source,
      mk_token("null"),
      SoftWare.Processes(gpid).Initialisation_State,
      mk_token("null"),
      mk_token("null"),
      false))
  else if is_Proxy(SoftWare.Processes(gpid)) then
  return(
    mk_Proxy(SoftWare.Processes(gpid).Source,
      SoftWare.Processes(gpid).Target,
      mk_Activity(SoftWare.Processes(gpid).Activity.Developer,
      SoftWare.Processes(gpid).Activity.Name,
      SoftWare.Processes(gpid).Activity.Source,
      mk_token("null"),
      SoftWare.Processes(gpid).Activity.Initialisation_State,
      mk_token("null"),
      mk_token("null"),
      false)))
  else
  return(
    mk_Duplex_Proxy(SoftWare.Processes(gpid).Source,
      SoftWare.Processes(gpid).Target1,
      SoftWare.Processes(gpid).Target2,
      mk_Activity(SoftWare.Processes(gpid).Activity.Developer,
      SoftWare.Processes(gpid).Activity.Name,
      SoftWare.Processes(gpid).Activity.Source,
      mk_token("null"),
      SoftWare.Processes(gpid).Activity.Initialisation_State,
      mk_token("null"),
      mk_token("null"),
      false)))
  pre CheckProcLoaded(SoftWare.Processes(gpid)) and
      gpid in set dom SoftWare.Processes;


ChangeProxySource1 : Global_Process_ID * Global_Process_ID *
Global_Process_ID ==> ()
ChangeProxySource1 (proxid, oldtarget, newtarget) ==
  (if is_Proxy(SoftWare.Processes(proxid)) then
     ChangeProxySource(proxid, newtarget);
   if is_Duplex_Proxy(SoftWare.Processes(proxid)) then
     ChangeDuplexProxySource(proxid, newtarget);
   if is_Condensing_Proxy(SoftWare.Processes(proxid)) then
     ChangeCondensingProxySource(proxid, oldtarget, newtarget);
  );


ChangeProxyTarget1 : Global_Process_ID * Global_Process_ID *
Global_Process_ID ==> ()
ChangeProxyTarget1 (proxid, oldtarget, newtarget) ==
  (if is_Proxy(SoftWare.Processes(proxid)) then
     ChangeProxyTarget(proxid, newtarget);
   if is_Duplex_Proxy(SoftWare.Processes(proxid)) then
     ChangeDuplexProxyTarget(proxid, oldtarget, newtarget);
   if is_Condensing_Proxy(SoftWare.Processes(proxid)) then
     ChangeCondensingProxyTarget(proxid, newtarget);
  );
```

```
AddProxy : Global_Process_ID * Global_Process_ID * Global_Process_ID * set
of Card_ID ==> ()
AddProxy (source, target, newpid, cids) ==
  (AddProc(newpid, mk_Proxy(source, target,
     mk_Activity(mk_token("Developer1"), mk_token("CORBA ORB"),
     mk_token("20. Goto 10"), mk_token("null"), mk_token("st1"),
     mk_token("null"), mk_token("null"), false)));
   AssignProcCards(newpid, settoseq(cids));
   LoadProc(newpid);
  );


AddDuplexProxy : Global_Process_ID * Global_Process_ID * Global_Process_ID *
Global_Process_ID * set of Card_ID ==> ()
AddDuplexProxy (source, targetpid1, targetpid2, newpid, cids) ==
  (AddProc(newpid, mk_Duplex_Proxy(source, targetpid1, targetpid2,
     mk_Activity(mk_token("Developer1"), mk_token("CORBA ORB"),
     mk_token("20. Goto 10"), mk_token("null"), mk_token("st1"),
     mk_token("null"), mk_token("null"), false)));
   AssignProcCards(newpid, settoseq(cids));
   LoadProc(newpid);
  );


AddCondensingProxy : Global_Process_ID * Global_Process_ID *
Global_Process_ID * Global_Process_ID * set of Card_ID ==> ()
AddCondensingProxy (source1, source2, target, newpid, cids) ==
  (AddProc(newpid, mk_Duplex_Proxy(source1, source2, target,
     mk_Activity(mk_token("Developer1"), mk_token("CORBA ORB"),
     mk_token("20. Goto 10"), mk_token("null"), mk_token("st1"),
     mk_token("null"), mk_token("null"), false)));
   AssignProcCards(newpid, settoseq(cids));
   LoadProc(newpid);
  );


AddLoc : Location * Loc_ID ==> ()
AddLoc(loc, locid) ==
  Loc.Physical_locations := Loc.Physical_locations munion {locid |-> loc}
  pre locid not in set dom Loc.Physical_locations;


RemoveLoc : Loc_ID ==> ()
RemoveLoc(locid) ==
  Loc.Physical_locations := {locid} <-: Loc.Physical_locations
  pre locid not in set rng HW_Loc_Map and
      locid in set dom Loc.Physical_locations;


ChangeLocName : Loc_ID * token ==> ()
ChangeLocName(locid, newname) ==
  Loc.Physical_locations(locid).Name := newname
  pre locid in set dom Loc.Physical_locations;


AssignMAULoc : MAU_ID * Loc_ID ==> ()
AssignMAULoc(mau, loc) ==
  HW_Loc_Map := HW_Loc_Map munion {mau |-> loc}
  pre loc in set dom Loc.Physical_locations and
      mau in set dom HardWare.MAUs and
      mau not in set dom HW_Loc_Map;


UnAssignMAULoc : MAU_ID ==> ()
UnAssignMAULoc(mau) ==
  HW_Loc_Map := {mau} <-: HW_Loc_Map
  pre mau in set dom HardWare.MAUs and
      mau in set dom HW_Loc_Map;
```

```
------------------------------
--Manipulate Model Operations--
--------Interesting Ops--------
------------------------------


LoadProc : Global_Process_ID ==> ()
LoadProc(gpid) ==
  if is_Activity(SoftWare.Processes(gpid))
  then
  SoftWare.Processes(gpid) :=
    mk_Activity(SoftWare.Processes(gpid).Developer,
      SoftWare.Processes(gpid).Name,
      SoftWare.Processes(gpid).Source,
      SoftWare.Processes(gpid).Initialisation_State,
      SoftWare.Processes(gpid).Initialisation_State,
      mk_token("initial"),
      mk_token("new PID"),
      true)
   else
   SoftWare.Processes(gpid).Activity :=
      mk_Activity(SoftWare.Processes(gpid).Activity.Developer,
      SoftWare.Processes(gpid).Activity.Name,
      SoftWare.Processes(gpid).Activity.Source,
      SoftWare.Processes(gpid).Activity.Initialisation_State,
      SoftWare.Processes(gpid).Activity.Initialisation_State,
      mk_token("initial"),
      mk_token("new PID"),
      true)
  pre gpid in set dom SW_HW_Map.Proc_to_Procrs and
      gpid in set dom SW_HW_Map.Proc_to_PMem and
      gpid in set dom SW_HW_Map.Proc_to_NPMem and
      CheckProcLoaded(SoftWare.Processes(gpid)) = false and
      gpid in set dom SoftWare.Processes;
--the pre condition checks that a process must have at least a processor,
some
--persistent memory, and some non-persistent memory.
--This makes the State of the process starting as a process will aquire
state
--when loaded


UnLoadProc : Global_Process_ID ==> ()
UnLoadProc(gpid) ==
  if is_Activity(SoftWare.Processes(gpid))
  then
  SoftWare.Processes(gpid) :=
    mk_Activity(SoftWare.Processes(gpid).Developer,
      SoftWare.Processes(gpid).Name,
      SoftWare.Processes(gpid).Source,
      mk_token("null"),
      SoftWare.Processes(gpid).Initialisation_State,
      mk_token("null"),
      mk_token("null"),
      false)
  else
  SoftWare.Processes(gpid).Activity :=
    mk_Activity(SoftWare.Processes(gpid).Activity.Developer,
      SoftWare.Processes(gpid).Activity.Name,
      SoftWare.Processes(gpid).Activity.Source,
      mk_token("null"),
      SoftWare.Processes(gpid).Activity.Initialisation_State,
      mk_token("null"),
      mk_token("null"),
      false)
  pre CheckProcLoaded(SoftWare.Processes(gpid)) and
      gpid in set dom SoftWare.Processes;
--This makes the State of the process null as a process that is not running
```

```
--should not have state


ChangeProcState : Global_Process_ID * token ==> ()
ChangeProcState(gpid, s) ==
  SoftWare.Processes(gpid).State := s
  pre CheckProcLoaded(SoftWare.Processes(gpid)) and
      gpid in set dom SoftWare.Processes;
--as processes do not really execute in VDM, this operation is provided to
--change the internal state of a process.


ChangeProcStack : Global_Process_ID * token ==> ()
ChangeProcStack(p, i) ==
  SoftWare.Processes(p).Instruction_Stack := i
  pre CheckProcLoaded(SoftWare.Processes(p)) and
      p in set dom SoftWare.Processes;
--as processes do not really execute in VDM, this operation is provided to
--change the internal instruction stack position.


CopyProcState : Global_Process_ID * Global_Process_ID ==> ()
CopyProcState(A, B) ==
  SoftWare.Processes(A).State := SoftWare.Processes(B).State
  pre A in set dom SoftWare.Processes and
      B in set dom SoftWare.Processes and
      CheckProcEquality(SoftWare.Processes(A), SoftWare.Processes(B)) and
      CheckIsRoute(A,B);


CopySDState : Shared_Data_ID * Shared_Data_ID ==> ()
CopySDState(A, B) ==
  SoftWare.SDs(A).State := SoftWare.SDs(B).State
  pre A in set dom SoftWare.SDs and
      B in set dom SoftWare.SDs and
      SoftWare.SDs(A).Protocol = SoftWare.SDs(B).Protocol and
      CheckCopySDState(A,B);
--This is abstract, as the processor must copy the state and this would
probably
--require a process on the processor being started and then being deleted
when
--done.


SynchProc : Global_Process_ID * Global_Process_ID ==> ()
SynchProc(A, B) ==
  SoftWare.Processes(A).Instruction_Stack :=
    SoftWare.Processes(B).Instruction_Stack
  pre A in set dom SoftWare.Processes and
      B in set dom SoftWare.Processes and
      CheckProcEquality(SoftWare.Processes(A), SoftWare.Processes(B)) and
      CheckIsRoute(A,B);
--this does not force either process to be running


ChangeLoadedProcID : Global_Process_ID * Global_Process_ID ==> ()
ChangeLoadedProcID(oldid, newid) ==
  (atomic(SoftWare.Processes := SoftWare.Processes munion {newid |->
SoftWare.Processes(oldid)};
   SW_HW_Map :=
     mk_SW_to_HW_Map(
       SW_HW_Map.Proc_to_Procrs munion
         {newid |-> SW_HW_Map.Proc_to_Procrs(oldid)},
       SW_HW_Map.Proc_to_PMem munion
         {newid |-> SW_HW_Map.Proc_to_PMem(oldid)},
       SW_HW_Map.Proc_to_NPMem munion
         {newid |-> SW_HW_Map.Proc_to_NPMem(oldid)},
       SW_HW_Map.SD_to_NPMem));
  RePointSWLinks(oldid, newid);
```

```
  RePointSDLinks(oldid, newid);
  atomic(SW_HW_Map :=
     mk_SW_to_HW_Map(
       {oldid} <-: SW_HW_Map.Proc_to_Procrs,
       {oldid} <-: SW_HW_Map.Proc_to_PMem,
       {oldid} <-: SW_HW_Map.Proc_to_NPMem,
       SW_HW_Map.SD_to_NPMem);
  SoftWare.Processes := {oldid} <-: SoftWare.Processes)
  )
pre oldid in set dom SoftWare.Processes and
    CheckProcLoaded(SoftWare.Processes(oldid)) and
    newid not in set dom SoftWare.Processes;
--Processes may need to change there ID especially when moving, thus this
--operation is provided.


MoveProcDelFirst : Global_Process_ID * seq of Card_ID ==> ()
MoveProcDelFirst(gpid, cids) ==
  (UnLoadProc(gpid);
   UnAssignProcCard(gpid);
   AssignProcCards(gpid, cids);
   LoadProc(gpid);
  )
  pre gpid in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(gpid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(gpid, cids);
--there is an implicit assumption here that a process can not really be
moved
--unless its loaded.  As its not really doing anything until loaded.  Checks
--that the move can be facilitated by the hardware links provided.


MoveProcDelAfter : Global_Process_ID * seq of Card_ID ==> ()
MoveProcDelAfter(gpid, cids) ==
  (AddProc(mk_token("temp_id"), UnloadedProc(gpid));
   AssignProcCards(mk_token("temp_id"), cids);
   LoadProc(mk_token("temp_id"));
   UnLoadProc(gpid);
   UnAssignProcCard(gpid);
   RePointSWLinks(gpid, mk_token("temp_id"));
   RePointSDLinks(gpid, mk_token("temp_id"));
   RemoveProc(gpid);
   ChangeLoadedProcID(mk_token("temp_id"), gpid);
  )
  pre gpid in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(gpid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(gpid, cids);
--there is an implicit assumption here that a process can not really be
moved
--unless its loaded.  As its not really doing anything until loaded.  Checks
--that the move can be facilitated by the hardware links provided.


MoveProcWState : Global_Process_ID * seq of Card_ID ==> ()
MoveProcWState(gpid, cids) ==
  (
   AddProc(mk_token("temp_id"), UnloadedProc(gpid));
   AssignProcCards(mk_token("temp_id"), cids);
   LoadProc(mk_token("temp_id"));
   CopyProcState(mk_token("temp_id"), gpid);
   UnLoadProc(gpid);
   UnAssignProcCard(gpid);
   RePointSWLinks(gpid, mk_token("temp_id"));
   RePointSDLinks(gpid, mk_token("temp_id"));
   RemoveProc(gpid);
   ChangeLoadedProcID(mk_token("temp_id"), gpid);
  )
```

```
  pre gpid in set dom SoftWare.Processes and
      mk_token("temp_id") not in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(gpid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(gpid, cids);
--there must be a point where the ID changes as both process cannot be
running
--with the same ID at the same time.  Checks that the move can be
facilitated
--by the hardware links provided.


MoveProcWStateAndSync : Global_Process_ID * seq of Card_ID ==> ()
MoveProcWStateAndSync(gpid, cids) ==
  (AddProc(mk_token("temp_id"), UnloadedProc(gpid));
   AssignProcCards(mk_token("temp_id"), cids);
   LoadProc(mk_token("temp_id"));
   CopyProcState(mk_token("temp_id"), gpid);
   SynchProc(mk_token("temp_id"), gpid);
   UnLoadProc(gpid);
   UnAssignProcCard(gpid);
   RePointSWLinks(gpid, mk_token("temp_id"));
   RePointSDLinks(gpid, mk_token("temp_id"));
   RemoveProc(gpid);
   ChangeLoadedProcID(mk_token("temp_id"), gpid);
  )
  pre gpid in set dom SoftWare.Processes and
      mk_token("temp_id") not in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(gpid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(gpid, cids);
--there must be a point where the ID changes as both process cannot be
running
--with the same ID at the same time.  Checks that the move can be
facilitated
--by the hardware links provided.


MoveProcDelFirstLP : Global_Process_ID * seq of Card_ID * set of
Global_Process_ID * set of Global_Process_ID ==> ()
MoveProcDelFirstLP(gpid, cids, sources, targets) ==
   def oldcids = SW_HW_Map.Proc_to_Procrs(gpid) union
     SW_HW_Map.Proc_to_NPMem(gpid) union SW_HW_Map.Proc_to_PMem(gpid) in
  (UnLoadProc(gpid);
   UnAssignProcCard(gpid);
   AssignProcCards(gpid, cids);
   LoadProc(gpid);
   for all s in set sources do
   (
     AddProxy(gpid, s, mk_token([gpid, card(rng SoftWare.Processes),
        s, card(dom SoftWare.Linkage)]), oldcids);
     ChangeProxySource1 (s, gpid, mk_token([gpid, card(rng
SoftWare.Processes)-1,
        s, card(dom SoftWare.Linkage)]));
   );
   for all t in set targets do
   (
     AddProxy(t, gpid, mk_token([gpid, card(rng SoftWare.Processes), t,
        card(dom SoftWare.Linkage)]), oldcids);
     ChangeProxyTarget1 (t, gpid, mk_token([gpid, card(rng
SoftWare.Processes)-1,
        t, card(dom SoftWare.Linkage)]))
   );
  )
  pre gpid in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(gpid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(gpid, cids);
--there is an implicit assumption here that a process cannot really be moved
```

```
--unless its loaded.  As its not really doing anything until loaded.  Checks
--that the move can be facilitated by the hardware links provided.


MoveProcDelAfterLP : Global_Process_ID * seq of Card_ID * set of
Global_Process_ID * set of Global_Process_ID ==> ()
MoveProcDelAfterLP(gpid, cids, sources, targets) ==
def oldcids = SW_HW_Map.Proc_to_Procrs(gpid) union
SW_HW_Map.Proc_to_NPMem(gpid)
    union SW_HW_Map.Proc_to_PMem(gpid) in
  (AddProc(mk_token("temp_id"), UnloadedProc(gpid));
   AssignProcCards(mk_token("temp_id"), cids);
   LoadProc(mk_token("temp_id"));
   UnLoadProc(gpid);
   UnAssignProcCard(gpid);
   RePointSWLinks(gpid, mk_token("temp_id"));
   RePointSDLinks(gpid, mk_token("temp_id"));
   RemoveProc(gpid);
   ChangeLoadedProcID(mk_token("temp_id"), gpid);
   for all s in set sources do
   (
     AddProxy(gpid, s, mk_token([gpid, card(rng SoftWare.Processes),
       s, card(dom SoftWare.Linkage)]), oldcids);
     ChangeProxySource1 (s, gpid, mk_token([gpid, card(rng
SoftWare.Processes)-1,
       s, card(dom SoftWare.Linkage)]));
   );
   for all t in set targets do
   (
     AddProxy(t, gpid, mk_token([gpid, card(rng SoftWare.Processes), t,
       card(dom SoftWare.Linkage)]), oldcids);
     ChangeProxyTarget1 (t, gpid, mk_token([gpid, card(rng
SoftWare.Processes)-1,
       t, card(dom SoftWare.Linkage)]));
   );
  )
  pre gpid in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(gpid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(gpid, cids);
--there is an implicit assumption here that a process cannot really be moved
--unless its loaded.  As its not really doing anything until loaded.  Checks
--that the move can be facilitated by the hardware links provided.


MoveProcWStateLP : Global_Process_ID * seq of Card_ID * set of
Global_Process_ID * set of Global_Process_ID ==> ()
MoveProcWStateLP(gpid, cids, sources, targets) ==
def oldcids = SW_HW_Map.Proc_to_Procrs(gpid) union
SW_HW_Map.Proc_to_NPMem(gpid)
       union SW_HW_Map.Proc_to_PMem(gpid) in
  (
   AddProc(mk_token("temp_id"), UnloadedProc(gpid));
   AssignProcCards(mk_token("temp_id"), cids);
   LoadProc(mk_token("temp_id"));
   CopyProcState(mk_token("temp_id"), gpid);
   UnLoadProc(gpid);
   UnAssignProcCard(gpid);
   RePointSWLinks(gpid, mk_token("temp_id"));
   RePointSDLinks(gpid, mk_token("temp_id"));
   RemoveProc(gpid);
   ChangeLoadedProcID(mk_token("temp_id"), gpid);
   for all s in set sources do
   (
     AddProxy(gpid, s, mk_token([gpid, card(rng SoftWare.Processes), s,
       card(dom SoftWare.Linkage)]), oldcids);
     ChangeProxySource1 (s, gpid, mk_token([gpid, card(rng
SoftWare.Processes)-1,
       s, card(dom SoftWare.Linkage)]));
```

```
   );
   for all t in set targets do
   (
     AddProxy(t, gpid, mk_token([gpid, card(rng SoftWare.Processes), t,
        card(dom SoftWare.Linkage)]), oldcids);
     ChangeProxyTarget1 (t, gpid, mk_token([gpid, card(rng
SoftWare.Processes)-1,
        t, card(dom SoftWare.Linkage)]))
   );
  )
  pre gpid in set dom SoftWare.Processes and
      mk_token("temp_id") not in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(gpid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(gpid, cids);
--there must be a point where the ID changes as both process cannot be
running
--with the same ID at the same time.  Checks that the move can be
facilitated
--by the hardware links provided.


MoveProcWStateAndSyncLP : Global_Process_ID * seq of Card_ID * set of
Global_Process_ID * set of Global_Process_ID ==> ()
MoveProcWStateAndSyncLP(gpid, cids, sources, targets) ==
def oldcids = SW_HW_Map.Proc_to_Procrs(gpid) union
SW_HW_Map.Proc_to_NPMem(gpid)
        union SW_HW_Map.Proc_to_PMem(gpid) in
  (AddProc(mk_token("temp_id"), UnloadedProc(gpid));
   AssignProcCards(mk_token("temp_id"), cids);
   LoadProc(mk_token("temp_id"));
   CopyProcState(mk_token("temp_id"), gpid);
   SynchProc(mk_token("temp_id"), gpid);
   UnLoadProc(gpid);
   UnAssignProcCard(gpid);
   RePointSWLinks(gpid, mk_token("temp_id"));
   RePointSDLinks(gpid, mk_token("temp_id"));
   RemoveProc(gpid);
   ChangeLoadedProcID(mk_token("temp_id"), gpid);
   for all s in set sources do
   (
     AddProxy(gpid, s, mk_token([gpid, card(rng SoftWare.Processes), s,
        card(dom SoftWare.Linkage)]), oldcids);
     ChangeProxySource1 (s, gpid, mk_token([gpid, card(rng
SoftWare.Processes)-1,
        s, card(dom SoftWare.Linkage)]));
   );
   for all t in set targets do
   (
     AddProxy(t, gpid, mk_token([gpid, card(rng SoftWare.Processes), t,
        card(dom SoftWare.Linkage)]), oldcids);
     ChangeProxyTarget1 (t, gpid, mk_token([gpid, card(rng
SoftWare.Processes)-1,
        t, card(dom SoftWare.Linkage)]))
   );
  )
  pre gpid in set dom SoftWare.Processes and
      mk_token("temp_id") not in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(gpid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(gpid, cids);
--there must be a point where the ID changes as both process cannot be
running
--with the same ID at the same time.  Checks that the move can be
facilitated
--by the hardware links provided.


CopyProc : Global_Process_ID * seq of Card_ID * Global_Process_ID ==> ()
```

```
CopyProc(existid, cids, newid) ==
  (AddProc(newid, UnloadedProc(existid));
   AssignProcCards(newid, cids);
   LoadProc(newid);
  )
  pre existid in set dom SoftWare.Processes and
      newid not in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(existid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(existid, cids);
--there is an implicit assumption here that a process cannot really be
copied
--unless its loaded.  As its not really doing anything until loaded.  Checks
--that the copy can be facilitated by the hardware links provided.


CopyProcWState : Global_Process_ID * seq of Card_ID * Global_Process_ID ==>
()
CopyProcWState(existid, cids, newid) ==
  (AddProc(newid, UnloadedProc(existid));
   AssignProcCards(newid, cids);
   LoadProc(newid);
   CopyProcState(newid, existid);
  )
  pre existid in set dom SoftWare.Processes and
      newid not in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(existid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(existid, cids);
--there must be a point where the ID changes as both process cannot be
running
--with the same ID at the same time.  Checks that the copy can be
facilitated
--by the hardware links provided.


CloneProc: Global_Process_ID * seq of Card_ID * Global_Process_ID==>()
CloneProc(existid, cids, newid) ==
  (AddProc(newid, UnloadedProc(existid));
   AssignProcCards(newid, cids);
   LoadProc(newid);
   CopyProcState(newid, existid);
   SynchProc(newid, existid);
  )
  pre existid in set dom SoftWare.Processes and
      newid not in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(existid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(existid, cids);
--there must be a point where the ID changes as both process cannot be
running
--with the same ID at the same time.  Checks that the clone can be
facilitated
--by the hardware links provided.


CopyProcWSWLinks : Global_Process_ID * seq of Card_ID * Global_Process_ID
==> ()
CopyProcWSWLinks(existid, cids, newid) ==
  (AddProc(newid, UnloadedProc(existid));
   AssignProcCards(newid, cids);
   LoadProc(newid);
   AddSameSWLinksAsProc(existid, newid);
  )
  pre existid in set dom SoftWare.Processes and
      newid not in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(existid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(existid, cids);
```

```
--there is an implicit assumption here that a process cannot really be
copied
--unless its loaded.  As its not really doing anything until loaded.  Checks
--that the copy can be facilitated by the hardware links provided.


CopyProcWStateAndSWLinks : Global_Process_ID * seq of Card_ID *
Global_Process_ID ==> ()
CopyProcWStateAndSWLinks(existid, cids, newid) ==
  (AddProc(newid, UnloadedProc(existid));
   AssignProcCards(newid, cids);
   LoadProc(newid);
   CopyProcState(newid, existid);
   AddSameSWLinksAsProc(existid, newid);
  )
  pre existid in set dom SoftWare.Processes and
      newid not in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(existid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(existid, cids);
--there must be a point where the ID changes as both process cannot be
running
--with the same ID at the same time.  Checks that the copy can be
facilitated
--by the hardware links provided.


CloneProcWSWLinks: Global_Process_ID * seq of Card_ID *
Global_Process_ID==>()
CloneProcWSWLinks(existid, cids, newid) ==
  (AddProc(newid, UnloadedProc(existid));
   AssignProcCards(newid, cids);
   LoadProc(newid);
   CopyProcState(newid, existid);
   SynchProc(newid, existid);
   AddSameSWLinksAsProc(existid, newid);
  )
  pre existid in set dom SoftWare.Processes and
      newid not in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(existid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(existid, cids);
--there must be a point where the ID changes as both process cannot be
running
--with the same ID at the same time.  Checks that the clone can be
facilitated
--by the hardware links provided.


CopyProcLP : Global_Process_ID * seq of Card_ID * Global_Process_ID * set of
Global_Process_ID * set of Global_Process_ID ==> ()
CopyProcLP(existid, cids, newid, sources, targets) ==
def oldcids = SW_HW_Map.Proc_to_Procrs(existid) union
      SW_HW_Map.Proc_to_NPMem(existid) union
SW_HW_Map.Proc_to_PMem(existid) in
  (AddProc(newid, UnloadedProc(existid));
   AssignProcCards(newid, cids);
   LoadProc(newid);
   for all s in set sources do
   (
     AddCondensingProxy(existid, newid, s, mk_token([existid,
       card(rng SoftWare.Processes), s, card(dom SoftWare.Linkage)]),
oldcids);
     ChangeProxySource1 (s, existid, mk_token([existid,
       card(rng SoftWare.Processes)-1, s, card(dom SoftWare.Linkage)]));
   );
   for all t in set targets do
   (
     AddDuplexProxy(t, existid, newid, mk_token([existid,
```

```
       card(rng SoftWare.Processes), t, card(dom SoftWare.Linkage)]),
oldcids);
     ChangeProxyTarget1 (t, existid, mk_token([existid,
       card(rng SoftWare.Processes)-1, t, card(dom SoftWare.Linkage)]))
   );
  )
  pre existid in set dom SoftWare.Processes and
      newid not in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(existid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(existid, cids);
--there is an implicit assumption here that a process cannot really be
copied
--unless its loaded.  As its not really doing anything until loaded.  Checks
--that the copy can be facilitated by the hardware links provided.


CopyProcWStateLP : Global_Process_ID * seq of Card_ID * Global_Process_ID *
set of Global_Process_ID * set of Global_Process_ID ==> ()
CopyProcWStateLP(existid, cids, newid, sources, targets) ==
def oldcids = SW_HW_Map.Proc_to_Procrs(existid) union
       SW_HW_Map.Proc_to_NPMem(existid) union
SW_HW_Map.Proc_to_PMem(existid) in
  (AddProc(newid, UnloadedProc(existid));
   AssignProcCards(newid, cids);
   LoadProc(newid);
   CopyProcState(newid, existid);
   for all s in set sources do
   (
     AddCondensingProxy(existid, newid, s, mk_token([existid,
       card(rng SoftWare.Processes), s, card(dom SoftWare.Linkage)]),
oldcids);
     ChangeProxySource1 (s, existid, mk_token([existid,
       card(rng SoftWare.Processes)-1, s, card(dom SoftWare.Linkage)]));
   );
   for all t in set targets do
   (
     AddDuplexProxy(t, existid, newid, mk_token([existid,
       card(rng SoftWare.Processes), t, card(dom SoftWare.Linkage)]),
oldcids);
     ChangeProxyTarget1 (t, existid, mk_token([existid,
       card(rng SoftWare.Processes)-1, t, card(dom SoftWare.Linkage)]))
   );
  )
  pre existid in set dom SoftWare.Processes and
      newid not in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(existid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(existid, cids);
--there must be a point where the ID changes as both process cannot be
running
--with the same ID at the same time.  Checks that the copy can be
facilitated
--by the hardware links provided.


CloneProcLP: Global_Process_ID * seq of Card_ID * Global_Process_ID * set of
Global_Process_ID * set of Global_Process_ID ==>()
CloneProcLP(existid, cids, newid, sources, targets) ==
def oldcids = SW_HW_Map.Proc_to_Procrs(existid) union
       SW_HW_Map.Proc_to_NPMem(existid) union
SW_HW_Map.Proc_to_PMem(existid) in
  (AddProc(newid, UnloadedProc(existid));
   AssignProcCards(newid, cids);
   LoadProc(newid);
   CopyProcState(newid, existid);
   SynchProc(newid, existid);
   for all s in set sources do
   (
```

```
     AddCondensingProxy(existid, newid, s, mk_token([existid,
       card(rng SoftWare.Processes), s, card(dom SoftWare.Linkage)]),
oldcids);
     ChangeProxySource1 (s, existid, mk_token([existid,
       card(rng SoftWare.Processes)-1, s, card(dom SoftWare.Linkage)]));
   );
   for all t in set targets do
   (
     AddDuplexProxy(t, existid, newid, mk_token([existid,
       card(rng SoftWare.Processes), t, card(dom SoftWare.Linkage)]),
oldcids);
     ChangeProxyTarget1 (t, existid, mk_token([existid,
       card(rng SoftWare.Processes)-1, t, card(dom SoftWare.Linkage)]))
   );
  )
  pre existid in set dom SoftWare.Processes and
      newid not in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(existid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(existid, cids);
--there must be a point where the ID changes as both process cannot be
running
--with the same ID at the same time.  Checks that the clone can be
facilitated
--by the hardware links provided.


CopyProcWSWLinksLP : Global_Process_ID * seq of Card_ID * Global_Process_ID
* set of Global_Process_ID * set of Global_Process_ID ==> ()
CopyProcWSWLinksLP(existid, cids, newid, sources, targets) ==
def oldcids = SW_HW_Map.Proc_to_Procrs(existid) union
       SW_HW_Map.Proc_to_NPMem(existid) union
SW_HW_Map.Proc_to_PMem(existid) in
  (AddProc(newid, UnloadedProc(existid));
   AssignProcCards(newid, cids);
   LoadProc(newid);
   AddSameSWLinksAsProc(existid, newid);
   for all s in set sources do
   (
     AddCondensingProxy(existid, newid, s, mk_token([existid,
       card(rng SoftWare.Processes), s, card(dom SoftWare.Linkage)]),
oldcids);
     ChangeProxySource1 (s, existid, mk_token([existid,
       card(rng SoftWare.Processes)-1, s, card(dom SoftWare.Linkage)]));
   );
   for all t in set targets do
   (
     AddDuplexProxy(t, existid, newid, mk_token([existid,
       card(rng SoftWare.Processes), t, card(dom SoftWare.Linkage)]),
oldcids);
     ChangeProxyTarget1 (t, existid, mk_token([existid,
       card(rng SoftWare.Processes)-1, t, card(dom SoftWare.Linkage)]))
   );
  )
  pre existid in set dom SoftWare.Processes and
      newid not in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(existid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(existid, cids);
--there is an implicit assumption here that a process cannot really be
copied
--unless its loaded.  As its not really doing anything until loaded.  Checks
--that the copy can be facilitated by the hardware links provided.


CopyProcWStateAndSWLinksLP : Global_Process_ID * seq of Card_ID *
Global_Process_ID * set of Global_Process_ID * set of Global_Process_ID ==>
()
CopyProcWStateAndSWLinksLP(existid, cids, newid, sources, targets) ==
```

```
def oldcids = SW_HW_Map.Proc_to_Procrs(existid) union
       SW_HW_Map.Proc_to_NPMem(existid) union
SW_HW_Map.Proc_to_PMem(existid) in
  (AddProc(newid, UnloadedProc(existid));
   AssignProcCards(newid, cids);
   LoadProc(newid);
   CopyProcState(newid, existid);
   AddSameSWLinksAsProc(existid, newid);
   for all s in set sources do
   (
     AddCondensingProxy(existid, newid, s, mk_token([existid,
       card(rng SoftWare.Processes), s, card(dom SoftWare.Linkage)]),
oldcids);
     ChangeProxySource1 (s, existid, mk_token([existid,
       card(rng SoftWare.Processes)-1, s, card(dom SoftWare.Linkage)]));
   );
   for all t in set targets do
   (
     AddDuplexProxy(t, existid, newid, mk_token([existid,
       card(rng SoftWare.Processes), t, card(dom SoftWare.Linkage)]),
oldcids);
     ChangeProxyTarget1 (t, existid, mk_token([existid,
       card(rng SoftWare.Processes)-1, t, card(dom SoftWare.Linkage)]))
   );
  )
  pre existid in set dom SoftWare.Processes and
      newid not in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(existid)) and
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(existid, cids);
--there must be a point where the ID changes as both process cannot be
running
--with the same ID at the same time.  Checks that the copy can be
facilitated
--by the hardware links provided.


CloneProcWSWLinksLP: Global_Process_ID * seq of Card_ID * Global_Process_ID
* set of Global_Process_ID * set of Global_Process_ID ==>()
CloneProcWSWLinksLP(existid, cids, newid, sources, targets) ==
def oldcids = SW_HW_Map.Proc_to_Procrs(existid) union
       SW_HW_Map.Proc_to_NPMem(existid) union
SW_HW_Map.Proc_to_PMem(existid) in
  (AddProc(newid, UnloadedProc(existid));
   AssignProcCards(newid, cids);
   LoadProc(newid);
   CopyProcState(newid, existid);
   SynchProc(newid, existid);
   AddSameSWLinksAsProc(existid, newid);
   for all s in set sources do
   (
     AddCondensingProxy(existid, newid, s, mk_token([existid,
       card(rng SoftWare.Processes), s, card(dom SoftWare.Linkage)]),
oldcids);
     ChangeProxySource1 (s, existid, mk_token([existid,
       card(rng SoftWare.Processes)-1, s, card(dom SoftWare.Linkage)]));
   );
   for all t in set targets do
   (
     AddDuplexProxy(t, existid, newid, mk_token([existid,
       card(rng SoftWare.Processes), t, card(dom SoftWare.Linkage)]),
oldcids);
     ChangeProxyTarget1 (t, existid, mk_token([existid,
       card(rng SoftWare.Processes)-1, t, card(dom SoftWare.Linkage)]))
   );
  )
  pre existid in set dom SoftWare.Processes and
      newid not in set dom SoftWare.Processes and
      CheckProcLoaded(SoftWare.Processes(existid)) and
```

```
      CheckRightCardTypes(cids) and
      CheckProcrsConnect(existid, cids);
--there must be a point where the ID changes as both process cannot be
running
--with the same ID at the same time.  Checks that the clone can be
facilitated
--by the hardware links provided.


CheckMAUSameLoc: MAU_ID * MAU_ID ==> bool
CheckMAUSameLoc(maul, mau2) ==
  return(HW_Loc_Map(maul) = HW_Loc_Map(mau2))
  pre maul in set dom HW_Loc_Map and
      mau2 in set dom HW_Loc_Map;
--if maus not allocated to a location, it is not known what the location of
the
--hardware is and thus this function cannot decide if they are in the same
--location.  More useful location functions could be added to check if two
--processes are in the same location etc...


--------------
--**VALUES**--
--------------


values


m1:MAU = mk_MAU(11, [<A>,<A>,<C>,<B>,<B>,<B>,<B>,<B>,<B>,<C>,<B>]);

mauid1:MAU_ID = mk_token("MAUID1");

m2:MAU = mk_MAU(5, [<A>,<A>,<C>,<B>,<B>]);

mauid2:MAU_ID = mk_token("MAUID2");

c1:Card = mk_Processor(mk_token("Man1"),mk_token("Mod1"),<A>, 1200, false,
            false, <CISC>, <MMX>, true, <x32Bit>, 10.0, 5.0, 10000.2, 50.0);

c2:Card = mk_Processor(mk_token("Man1"),mk_token("Mod1"),<A>, 1200, false,
            false, <CISC>, <MMX>, true, <x32Bit>, 10.0, 5.0, 10000.2, 50.0);

c3:Card = mk_Processor(mk_token("Man1"),mk_token("Mod1"),<B>, 1200, false,
            false, <CISC>, <MMX>, true, <x32Bit>, 10.0, 5.0, 10000.2, 50.0);

c4:Card = mk_Persistent_Mem(mk_token("Man2"),mk_token("Mod2"),<B>, 24, 50,
12,
            false, 24, 50, 12, 20000, 100, -10, 5040.6, 50.0);

c5:Card = mk_Persistent_Mem(mk_token("Man2"),mk_token("Mod2"),<B>, 24, 50,
12,
            false, 24, 50, 12, 20000, 100, -10, 5040.6, 50.0);

c6:Card = mk_Non_Persistent_Mem(mk_token("Man3"),mk_token("Mod3"),<B>, 24,
50,
            12, 256, 333, 100, -10, 4096.6, 50.0);

c7:Card = mk_Non_Persistent_Mem(mk_token("Man3"),mk_token("Mod3"),<B>, 24,
50,
            12, 256, 333, 100, -10, 4096.6, 50.0);

c8:Card = mk_Non_Persistent_Mem(mk_token("Man3"),mk_token("Mod3"),<B>, 24,
50,
            12, 512, 400, 100, -10, 4096.6, 50.0);

c9:Card = mk_Processor(mk_token("Man1"),mk_token("Mod2"),<C>, 1200, false,
            false, <CISC>, <MMX>, true, <x32Bit>, 10.0, 5.0, 10000.2, 50.0);
```

```
c10:Card = mk_Processor(mk_token("Man1"),mk_token("Mod2"),<C>, 1200, false,
           false, <CISC>, <MMX>, true, <x32Bit>, 10.0, 5.0, 10000.2, 50.0);

c11:Card = mk_Non_Persistent_Mem(mk_token("Man3"),mk_token("Mod3"),<B>, 24,
50,
           12, 512, 400, 100, -10, 4096.6, 50.0);

cardid1:Card_ID = mk_token("CardID1");

cardid2:Card_ID = mk_token("CardID2");

cardid3:Card_ID = mk_token("CardID3");

cardid4:Card_ID = mk_token("CardID4");

cardid5:Card_ID = mk_token("CardID5");

cardid6:Card_ID = mk_token("CardID6");

cardid7:Card_ID = mk_token("CardID7");

cardid8:Card_ID = mk_token("CardID8");

cardid9:Card_ID = mk_token("CardID9");

cardid10:Card_ID = mk_token("CardID10");

cardid11:Card_ID = mk_token("CardID11");

procid1:Global_Process_ID = mk_token("ProcID1");

procid2:Global_Process_ID = mk_token("ProcID2");

procid3:Global_Process_ID = mk_token("ProcID3");

procid4:Global_Process_ID = mk_token("ProcID4");

procid5:Global_Process_ID = mk_token("ProcID5");

procid6:Global_Process_ID = mk_token("ProcID6");

procid7:Global_Process_ID = mk_token("ProcID7");

procid8:Global_Process_ID = mk_token("ProcID8");

p1:Process = mk_Activity(mk_token("Developer1"), mk_token("CORBA ORB"),
             mk_token("20. Goto 10"), mk_token("null"), mk_token("st1"),
             mk_token("null"), mk_token("null"), false);

p2:Process = mk_Activity(mk_token("Developer1"), mk_token("CORBA ORB"),
             mk_token("20. Goto 10"), mk_token("null"), mk_token("st1"),
             mk_token("null"), mk_token("null"), false);

p3:Process = mk_Activity(mk_token("Developer3"), mk_token("FIFA"),
             mk_token("20. Printf"), mk_token("null"),  mk_token("st2"),
             mk_token("null"), mk_token("null"), false);

p4:Process = mk_Proxy(procid1, procid5, p1);

p5:Process = mk_Proxy(procid4, procid2, p1);

sdid1 = mk_token("SD1");

sdid2 = mk_token("SD2");

sd1:Shared_Data = mk_Shared_Data(<Pool>, mk_token("Stateb"));

sd2:Shared_Data = mk_Shared_Data(<Pool>, mk_token("Statec"));
```

```
linkid1:Link_ID = mk_token("Link_ID");
linkid2:Link_ID = mk_token("LinkID2");
linkid3:Link_ID = mk_token("LinkID3");
linkid4:Link_ID = mk_token("LinkID4");
linkid5:Link_ID = mk_token("LinkID5");
linkid6:Link_ID = mk_token("LinkID6");
linkid7:Link_ID = mk_token("LinkID7");
linkid8:Link_ID = mk_token("LinkID8");
linkid9:Link_ID = mk_token("LinkID9");
linkid10:Link_ID = mk_token("LinkID10");
linkid11:Link_ID = mk_token("LinkID11");
linkid12:Link_ID = mk_token("LinkID12");
linkid13:Link_ID = mk_token("LinkID13");
linkid14:Link_ID = mk_token("LinkID14");
linkid15:Link_ID = mk_token("LinkID15");
linkid16:Link_ID = mk_token("LinkID16");
linkid17:Link_ID = mk_token("LinkID17");
linkid18:Link_ID = mk_token("LinkID18");
linkid19:Link_ID = mk_token("LinkID19");
linkid20:Link_ID = mk_token("LinkID20");
linkid21:Link_ID = mk_token("LinkID21");
linkid22:Link_ID = mk_token("LinkID22");
linkid23:Link_ID = mk_token("LinkID23");
linkid24:Link_ID = mk_token("LinkID24");
linkid25:Link_ID = mk_token("LinkID25");
linkid26:Link_ID = mk_token("LinkID26");
servid1:Service_ID = mk_token("service1");
servid2:Service_ID = mk_token("service2");
servid3:Service_ID = mk_token("service3");
locid1:Loc_ID = mk_token("location 1");
locid2:Loc_ID = mk_token("location 2");
locid3:Loc_ID = mk_token("location 3");
location1:Location = mk_Location(mk_token("Newcastle"), 12.2, 6.5);
location2:Location = mk_Location(mk_token("Manchester"), 24.2, 46.5);
location3:Location = mk_Location(mk_token("Colchester"), 33.2, 16.99);
```

BLANK PAGE IN ORIGINAL

Appendix B

CSP Thrashing Definitions

This appendix contains the full timed and un-timed CSP configuration thrashing models introduced in chapter 3 of this thesis. The two models are presented in the two subsections below.

1. *Un-timed CSP Configuration Thrashing Model*

```
channel move, startup, doa, overlap, start_min_wk, end_min_wk, thrashbang

MONITOR = startup ->
               (start_min_wk -> move -> overlap -> MONITOR
                [] start_min_wk -> end_min_wk -> move -> MONITOR
                [] move -> overlap -> MONITOR)
          []move -> overlap -> MONITOR

PROCESS = startup ->
               (start_min_wk -> move -> PROCESS
                [] start_min_wk -> end_min_wk -> doa -> move -> PROCESS)
          []move -> PROCESS

PROCESSNT = startup ->
               (start_min_wk -> move -> startup -> start_min_wk ->
end_min_wk -> doa -> move -> PROCESSNT
                [] start_min_wk -> end_min_wk -> doa -> move -> PROCESSNT)
             []move -> startup -> start_min_wk -> end_min_wk -> move ->
PROCESSNT

SYSTEM =
(MONITOR|[{startup,move,start_min_wk,end_min_wk}]|(PROCESS\{doa}))\{startup,
start_min_wk,move}

SYSTEMNT =
(MONITOR|[{startup,move,start_min_wk,end_min_wk}]|(PROCESSNT\{doa}))\{startu
p,start_min_wk,move}

TEST = (SYSTEM|[{overlap, end_min_wk}]|THRASH(3,3))\{overlap, end_min_wk}

TEST1 = (SYSTEMNT|[{overlap, end_min_wk}]|THRASH(3,3))\{overlap, end_min_wk}

THRASH(max, x) = if (x==0) then
                   thrashbang -> STOP
                 else
                   overlap -> THRASH(max, x-1)
                   []end_min_wk->THRASH(max,max)

assert STOP [T= TEST
assert STOP [T= TEST1
```

2. *Timed CSP Configuration Thrashing Model*

```
T = 3
```

```
O = 1

channel move, startup, doa, overlap, start_min_wk, end_min_wk, tock,
thrashbang

TOCKS = tock -> TOCKS

MONITOR = startup ->
               (start_min_wk -> move -> overlap -> MONITOR
                [] start_min_wk -> end_min_wk -> move -> MONITOR
                [] move -> overlap -> MONITOR)
          []move -> overlap -> MONITOR

PROCESS = startup -> tock ->
               (start_min_wk -> tock ->  move -> tock -> PROCESS
                [] start_min_wk -> tock -> end_min_wk -> tock -> doa -> tock
-> move -> tock -> PROCESS)
          []move -> tock -> PROCESS

PROCESSNT = startup -> tock ->
               (start_min_wk -> tock ->  move -> tock -> startup -> tock ->
start_min_wk -> tock -> end_min_wk -> tock -> doa -> tock -> move -> tock ->
PROCESSNT
                [] start_min_wk -> tock -> end_min_wk -> tock -> doa -> tock
-> move -> tock -> PROCESSNT)
            []move -> tock -> startup -> tock -> start_min_wk -> tock ->
end_min_wk -> tock -> move -> tock -> PROCESSNT

SYSTEM =
(MONITOR|[{startup,move,start_min_wk,end_min_wk}]|(PROCESS\{doa}))\{end_min_
wk,startup,start_min_wk,move}|[{tock}]|TOCKS

SYSTEMNT =
(MONITOR|[{startup,move,start_min_wk,end_min_wk}]|(PROCESSNT\{doa}))\{end_mi
n_wk, startup,start_min_wk,move}|[{tock}]|TOCKS

THRASHTIMED(<>, maxt, maxo) = overlap -> THRASHTIMED(<>^<O>,maxt,maxo)
                  [] tock -> THRASHTIMED(<>^<T>,maxt,maxo)

THRASHTIMED(x, maxt, maxo) = if (numo(x) == maxo) then
                     thrashbang -> STOP
                   else
                     overlap -> THRASHTIMED(x^<O>,maxt,maxo)
                     [] tock -> if (numt(x) == maxt) then
                          THRASHTIMED(slidewindow(x)^<T>,maxt,maxo)
                        else
                          THRASHTIMED(x^<T>,maxt,maxo)

numt(x) = length(<y | y <- x, y==T>)

numo(x) = length(<y | y <- x, y==O>)

slidewindow(x) = if (head(x) == T) then
                   tail(x)
                 else
                   slidewindow(tail(x))

TEST = (SYSTEM|[{overlap, tock}]|THRASHTIMED(<>,2,3))\{tock,overlap}

TEST1 = (SYSTEMNT|[{overlap, tock}]|THRASHTIMED(<>,2,3))\{tock,overlap}

assert STOP [T= TEST
assert STOP [T= TEST1
```

Appendix C

Possible Process Requirements

This appendix presents possible processor, memory, OS, and storage requirements for processes. All of the requirements outlined are attributes a process could require in order to function. The attributes outlined in this appendix are only a candidate set of attributes.

1. Possible Processor Requirements

- Make (e.g. intel, AMD)
- Model (e.g. P4, Athlon)
- Cpu speed (e.g. 1200MHz)
- Cache size
    - L1 cache size
    - L2 cache size
    - L3 cache size
- Pipelining
- Operating voltage
- Bus frequency (MHz)
- Number of channels
- Core frequency (MHz)
- Bus/Core ratio
- Instruction sets (e.g. 3D NOW etc....)
- Max operational temperature
- Min operational temperature
- Dimensions (size)
- Number of transistors (Complexity – new AMD 64 processors have 105.9million – more chance of failure?)
- Co-processor
- Register size (e.g. 32bit, 64bit – particularly makes a difference to amount of physical memory can be accessed, i.e. number addresses held)
- Socket type (e.g. Socket 7)
- Cycle time
- Mics (the size of the line widths in microns of the microchip process that the microchip is built on.)
- Integrated Floating Point Unit (FPU)
- Max operational altitude
- Allotment of time needed on processor

- Max humidity

2. Possible Memory Requirements

- Make (e.g. Kingston, IBM)
- Type (e.g. DDR, SDRAM)
- Speed / Bus clock rate (e.g. 200MHz, 400MHz)
- Size (e.g. 128mb, 256mb)
- Socket type (e.g. DIMM 184pin)
- Number of chips
- Operating voltage
- Dimensions (size)
- Max operational temperature
- Min operational temperature
- Max operational altitude

3. Possible Operating System (OS) Requirements

- Developer (e.g. Microsoft)
- O.S. Name (e.g. Linux, Windows, etc....)
- Single threaded / multi-threading
- Scheduling algorithms (may need to specify one if not coded in process)
- Processor support (support for co-processor, instruction sets etc...)
- Memory support (support for the memory used)
    - Memory management (management of the pool of memory)
- Fault tolerance support (code to support FT activities)
- Non interference support (guarantees from the OS that other processes will not interfere – this is generally kernel verification guarantees)
- Hardware transparency support
- Network support
- Number of lines of code (complexity of OS)

4. Possible Storage Requirements

- Max humidity

- Max operational temperature
- Min operational temperature
- Max operational altitude
- Weight of storage equipment
- Storage capacity
- Cache size
- Seek time
    - Average
    - Worst case
    - Best case
- Data transfer rate
    - Average
    - Worst case
    - Best case
- Dimensions of storage equipment

Appendix D

Demonstrator Java Source Code

This appendix presents the source code for the reconfigurable systems demonstrator. The demonstrator outlined has been built using Java RMI. Java RMI provides a means of invoking methods on remote Java objects. Each class and java file is presented in a separate subsection.

1. CTEST.java

```
//TEST JAVA FILE
//Role: Locates a controller and creates a Reconfigurable Process
//      and increments its value

import
jcode.process.ReconfigProcess;
import jcode.controller.Controller;
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;


public class CTEST
{
  public static void main(String[] args)
  {
    try
    {
      if (args.length != 1)
      {
        System.out.println("Wrong Arguments - Try Again");
        return;
      }
      System.out.println("Starting Test");
      Controller cont = (Controller)
Naming.lookup("rmi://localhost:1099/Controller".concat(args[0]));
      System.out.println("controller reference obtained");
      ReconfigProcess p = cont.createProcess("proc1");
      System.out.println(p.getData());
      p.compute();
      System.out.println(p.getData());
    }
    catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }
}
```

2. CTEST1.java

```
//TEST JAVA FILE
//Role: Locates Reconfigurable Process Proc1 and increments it value

import jcode.process.ReconfigProcess;
import jcode.controller.Controller;
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
```

```
public class CTEST1
{
  public static void main(String[] args)
  {
    try
    {
      System.out.println("Starting Test");
      ReconfigProcess p = (ReconfigProcess)
Naming.lookup("rmi://localhost:1099/proc1");
      System.out.println("ReconfigProcess reference obtained");
      System.out.println(p.getData());
      p.compute();
      System.out.println(p.getData());
    }
    catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }
}
```

3. CTEST2.java

```
//TEST JAVA FILE
//Role: Locates Reconfigurable Process Proc1 and Moves it
//      using MoveProcDelFirst to localhost

import jcode.process.ReconfigProcess;
import jcode.controller.Controller;
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;


public class CTEST2
{
  public static void main(String[] args)
  {
    try
    {
      if (args.length != 1)
      {
        System.out.println("Wrong Arguments - Try Again");
        return;
      }
      System.out.println("Starting Test");
      Controller cont = (Controller)
Naming.lookup("rmi://localhost:1099/Controller".concat(args[0]));
      System.out.println("controller reference obtained");
      cont.MoveProcDelFirst("proc1","localhost");
    }
    catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }
}
```

4. CTEST3.java

```
//TEST JAVA FILE
//Role: Locates Reconfigurable Process Proc1 and Moves it
//      using MoveProcDelAfter to localhost
```

```
import jcode.process.ReconfigProcess;
import jcode.controller.Controller;
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;


public class CTEST3
{
  public static void main(String[] args)
  {
    try
    {
      if (args.length != 1)
      {
        System.out.println("Wrong Arguments - Try Again");
        return;
      }
      System.out.println("Starting Test");
      Controller cont = (Controller)
Naming.lookup("rmi://localhost:1099/Controller".concat(args[0]));
      System.out.println("controller reference obtained");
      cont.MoveProcDelAfter("proc1","localhost");
    }
    catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }
}
```

5. CTEST4.java

```
//TEST JAVA FILE
//Role: Locates Reconfigurable Process Proc1 and Moves it
//      using MoveProcWState to localhost

import jcode.process.ReconfigProcess;
import jcode.controller.Controller;
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;


public class CTEST4
{
  public static void main(String[] args)
  {
    try
    {
      if (args.length != 1)
      {
        System.out.println("Wrong Arguments - Try Again");
        return;
      }
      System.out.println("Starting Test");
      Controller cont = (Controller)
Naming.lookup("rmi://localhost:1099/Controller".concat(args[0]));
      System.out.println("controller reference obtained");
      cont.MoveProcWState("proc1","localhost");
    }
    catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }
```

```
}

//TEST JAVA FILE
//Role: Locates Controller and Adds Rules for proc1
//      also enables rule checking

import jcode.process.ReconfigProcess;
import jcode.controller.Controller;
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
```

6. CTEST5.java

```
public class CTEST5
{
  public static void main(String[] args)
  {
    try
    {
      if (args.length != 1)
      {
        System.out.println("Wrong Arguments - Try Again");
        return;
      }
      System.out.println("Starting Test");
      Controller cont = (Controller)
Naming.lookup("rmi://localhost:1099/Controller".concat(args[0]));
      System.out.println("controller reference obtained");
      cont.EnableRules();
      Object[] x = new Object[5];
      x[0] = "ConfOver";
      x[1] = "proc1";
      x[2] = new Integer(2);
      x[3] = new Integer(170);
      x[4] = new Integer(51);
      cont.AddRule(x);
    }
    catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }
}
```

7. ControllerImpl.java

```
package jcode.controller;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

import java.rmi.server.UnicastRemoteObject;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.util.LinkedList;
import java.util.Date;
import jcode.process.*;
import jcode.factory.*;

class ControllerImpl extends UnicastRemoteObject implements Controller
{

  int rules = 0;
  LinkedList RuleStore = new LinkedList();

  //History stored as Longs in the LinkedList
```

```
  //This does not allow for control of multiple
  //processes - though alterations to the code
  //to provide a LinkedList for each process
  //would allow this.  This code was not needed
  //for the demonstrator.
  LinkedList History = new LinkedList();

  //Constructor
  public ControllerImpl() throws RemoteException
  {
    super();
  }

  //Main - program creates its own RMI registry
  public static void main(String[] args)
  {
    try
    {
      System.out.println("Controller started");
      String a = new String("Controller");
      String cn = a.concat(args[0]);
      LocateRegistry.getRegistry().bind(cn, new ControllerImpl());
      System.out.println("Controller bound to localhost as ".concat(cn));
    } catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }

  public void EnableRules() throws RemoteException
  {
    this.rules=1;
  }

  public void DisableRules() throws RemoteException
  {
    this.rules=0;
  }

  private int ReconfigYN(String RMIname)
  {
    if (this.rules == 0)
    {
      //rules not applicable thus reconfiguration fine
      System.out.println("Rules Not Applicable - Reconfiguration
Permitted");
      return 1;
    }
    else
    {
      //need to check rules
      if (RuleStore.size() == 0)
      {
        System.out.println("No Rules Exist");
        return 1;
      }
      for (int i = 0;i<RuleStore.size(); i++)
      {
        Object[] Rule = (Object[])RuleStore.get(i);
        //rules with knock on effects like collocated should be added twice
        //to simplify checks
        if (((String)Rule[1]).compareTo(RMIname) == 0)
        {
          System.out.println("FOUND A RULE THAT APPLIES");
          //only one type of rule presently available
          if (((String)Rule[0]).compareTo("ConfOver") == 0)
          {
            int x = ((Integer)Rule[2]).intValue();
```

```
            int y = ((Integer)Rule[3]).intValue();
            int z = ((Integer)Rule[4]).intValue();

            Date date = new Date();
            long currentTime = date.getTime();
            long lastReconfig = 0;

            if (History.size() == 0)
            {
              System.out.println("History Empty - Not an overlap");
              History.add(new Long(currentTime));
              return 1;
            }
            else
            {
              lastReconfig = ((Long)History.getLast()).longValue();
              if ((currentTime - lastReconfig) > ((long)z)*1000)
              {
                System.out.println("Not an overlap");
                History.add(new Long(currentTime));
                return 1;
              }
              else
              {
                int count = 0;
                long stInterval = currentTime - (y*1000);
                for (int a=0; a<History.size(); a++)
                {
                  if (((Long)History.get(a)).longValue() > stInterval)
                  {
                    System.out.println("ADD ONE");
                    count++;
                  }
                }
                if (count < x)
                {
                  System.out.println("Overlap Occured");
                  History.add(new Long(currentTime));
                  return 1;
                }
              }
            }
            //Need overlaps in y interval
            System.out.println("Reconfiguration Denied");
            return 0;
          }
        }
      }
      return 1;
    }
  }

  public void AddRule(Object[] newrule) throws RemoteException
  {
    //for live system checks for rule well formedness would be required
    this.RuleStore.add(newrule);
  }

  public void RemoveRule(Object[] remrule) throws RemoteException
  {
    if (this.RuleStore.contains(remrule))
    {
      this.RuleStore.remove(remrule);
    }
    else
    {
      System.out.println("Rule Does not Exist");
    }
  }
```

```
  public ReconfigProcess createProcess(String ProcName) throws
RemoteException
  {
    try
    {
      System.out.println("Trying to Create Process");
      ProcessFactory fact = (ProcessFactory)
Naming.lookup("rmi://localhost:1099/ProcessFactory");
      ReconfigProcess a = fact.createProcess(ProcName);
      System.out.println("New Process Created");
      this.bindProcess(a, "localhost", ProcName);
      return a;
    }catch(Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
      throw new RemoteException("Process Cannot Be Created");
    }
  }

  //Function to enable del after functions - multiple binds not possible in
RMIRegistry
  private ReconfigProcess createProcNoBind(String ProcName) throws
RemoteException
  {
    try
    {
      System.out.println("Trying to Create Process");
      ProcessFactory fact = (ProcessFactory)
Naming.lookup("rmi://localhost:1099/ProcessFactory");
      ReconfigProcess a = fact.createProcess(ProcName);
      System.out.println("New Process Created");
      //this.bindProcess(a, "localhost");
      return a;
    }catch(Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
      throw new RemoteException("Process Cannot Be Created");
    }
  }

  public void bindProcess(ReconfigProcess a, String loc, String ProcName)
throws RemoteException
  {
    try
    {
      String cn = ProcName;
      LocateRegistry.getRegistry(loc).bind(cn, a);
      System.out.println("ReconfigProcess bound to".concat(loc.concat(" as
".concat(cn))));
    }catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }

  public void MoveProcDelFirst(String RMIname, String newloc) throws
RemoteException
  {
    //Check Reconfiguration can occur
    if (this.ReconfigYN(RMIname) == 1)
    {
      try
      {
        ReconfigProcess p = (ReconfigProcess)
Naming.lookup("rmi://localhost:1099/".concat(RMIname));
```

```
        System.out.println("ReconfigProcess reference obtained");
        p.delProcess();
        System.out.println("Process Deleted");
        ReconfigProcess p1 = this.createProcess(RMIname);
        System.out.println("New Process Created");
      }catch (Exception e)
      {
        System.out.println("Exception caught: " + e);
        e.printStackTrace();
      }
    }
    else
    {
      System.out.println("Reconfiguration Denied");
    }
  }

  public void MoveProcDelAfter(String RMIname, String newloc) throws
RemoteException
  {
    if (this.ReconfigYN(RMIname) == 1)
    {
      try
      {
        ReconfigProcess p1 = this.createProcNoBind(RMIname);
        System.out.println("New Process Created");
        ReconfigProcess p = (ReconfigProcess)
Naming.lookup("rmi://localhost:1099/".concat(RMIname));
        System.out.println("ReconfigProcess reference obtained");
        p.delProcess();
        System.out.println("Process Deleted");
        //bind must take place last as RMIRegistry does not allow
        //multiple binds to same name.
        this.bindProcess(p1, "localhost", RMIname);
      }catch (Exception e)
      {
        System.out.println("Exception caught: " + e);
        e.printStackTrace();
      }
    }
    else
    {
      System.out.println("Reconfiguration Denied");
    }
  }

  public void MoveProcWState(String RMIname, String newloc) throws
RemoteException
  {
    if (this.ReconfigYN(RMIname) == 1)
    {
      try
      {
        ReconfigProcess p1 = this.createProcNoBind(RMIname);
        System.out.println("New Process Created");
        ReconfigProcess p = (ReconfigProcess)
Naming.lookup("rmi://localhost:1099/".concat(RMIname));
        System.out.println("ReconfigProcess reference obtained");
        p1.setData(p.getData());
        System.out.println("Process State Transfered");
        p.delProcess();
        System.out.println("Process Deleted");
        //bind must take place last as RMIRegistry does not allow
        //multiple binds to same name.
        this.bindProcess(p1, "localhost", RMIname);
      }catch (Exception e)
      {
        System.out.println("Exception caught: " + e);
        e.printStackTrace();
```

```
      }
    }
    else
    {
      System.out.println("Reconfiguration Denied");
    }
  }

}
```

8. Controller.java

```
package jcode.controller;
import jcode.process.*;
import java.rmi.*;

public interface Controller extends Remote
{
  public ReconfigProcess createProcess(String ProcName) throws
RemoteException;
  public void bindProcess(ReconfigProcess a, String newloc, String ProcName)
throws RemoteException;
  public void MoveProcDelFirst(String RMIname, String newloc) throws
RemoteException;
  public void MoveProcDelAfter(String RMIname, String newloc) throws
RemoteException;
  public void MoveProcWState(String RMIname, String newloc) throws
RemoteException;
  public void EnableRules() throws RemoteException;
  public void DisableRules() throws RemoteException;
  public void AddRule(Object[] newrule) throws RemoteException;
  public void RemoveRule(Object[] remrule) throws RemoteException;
}
```

9. ProcessFactoryImpl.java

```
package jcode.factory;
//import java.rmi.*;
//import java.rmi.server.*;
//import java.rmi.registry.*;

import java.rmi.server.UnicastRemoteObject;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import jcode.process.*;

//public class AgentFactoryImpl extends UnicastRemoteObject implements
AgentFactory
class ProcessFactoryImpl extends UnicastRemoteObject implements
ProcessFactory
{
  int data = 0;

  //Constructor
  public ProcessFactoryImpl() throws RemoteException
  {
    super();
  }

  //Main - program creates its own RMI registry
  public static void main(String[] argv)
  {
    try
    {
      System.out.println("Process factory started");
      //LocateRegistry.createRegistry(1099);
      //System.out.println("Registry started");
      //registry must be started
```

```
      LocateRegistry.getRegistry().bind("ProcessFactory",new
ProcessFactoryImpl());
      System.out.println("Process factory bound");
    } catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }

  public ReconfigProcess createProcess(String ProcName) throws
RemoteException
  {
    ReconfigProcessImpl a = new ReconfigProcessImpl(ProcName);
    //a.setData(this.data);
    return (ReconfigProcess)java.rmi.server.RemoteObject.toStub(a) ;
  }

  public void setProcess(ReconfigProcess a) throws RemoteException
  {
    this.data += a.getData();
  }
}
```

10. ProcessFactory.java

```
package jcode.factory;
import jcode.process.*;
import java.rmi.*;

public interface ProcessFactory extends Remote
{
  public ReconfigProcess createProcess(String ProcName)    throws
RemoteException;
  public void setProcess(ReconfigProcess a) throws RemoteException;
}
```

11. ReconfigProcessImpl.java

```
package jcode.process;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

import java.rmi.server.UnicastRemoteObject;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import jcode.process.*;
import jcode.factory.*;


public class ReconfigProcessImpl extends UnicastRemoteObject implements
ReconfigProcess
{
  String bindname;
  int data;
  MainThread mt = new MainThread();

  //Constructor
  public ReconfigProcessImpl(String bindname) throws RemoteException
  {
    super();
    this.bindname = bindname;
    System.out.println("ReconfigProcess Initialising");
    mt.start();
  }

  //No Main As Process is not meant to run from command line
```

```
  public void delProcess() throws RemoteException
  {
    //No need to call a deconstructor as Java RMI uses a Distributed Garbage
Collection
    //system and thus will remove stale processes once no reference exists
for them.
    try {
      String cn = this.bindname;
      System.out.println("ReconfigProcess Uninitialising");
      LocateRegistry.getRegistry().unbind(cn);
      mt.stop();
      System.out.println("ReconfigProcess no longer bound to localhost as
".concat(cn));
    }catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }

  public void compute() throws RemoteException
  {
    data += 10;
  }

  public int getData() throws RemoteException
  {
    return data;
  }

  public void setData(int data) throws RemoteException
  {
    this.data = data;
  }
}
```

12. ReconfigProcess.java

```
package jcode.process;
import java.rmi.*;
import jcode.process.*;
import jcode.factory.*;


public interface ReconfigProcess extends Remote
{
  public void delProcess() throws RemoteException;
  public void compute() throws RemoteException;
  public int getData() throws RemoteException;
  public void setData(int data) throws RemoteException;
}
```

13. MainThread.java

```
package jcode.process;

public class MainThread extends Thread
{
  public void run()
  {
    int i = 0;
    for (i=0; i<10; i++)
    {
      try
      {
        Thread.sleep(5000);
      }catch (Exception e)
      {}
```

```
      System.out.println(i);
    }
    System.out.println("cycle complete");
  }
}
```

Appendix E

Case Study

This appendix contains the full CSP configuration thrashing models used for the case study, as well as the java source code used for the software approach to the case study. The models and java classes are presented in subsections below.

As well as the models and java classes this appendix contains the output from an unconstrained java system and a contained java system.

1. Un-timed CSP Case Study Model

```
T = 3

O = 1

channel move, startup, output_decision, send_to_fusion, start_fuse_data,
end_fuse_data, send_signal, recieve_signal, start_process_data,
end_process_data, overlap, start_min_wk, end_min_wk, tock, thrashbang

TOCKS = tock -> TOCKS

MONITOR = startup ->
                (start_min_wk -> move -> overlap -> MONITOR
                 [] start_min_wk -> end_min_wk -> move -> MONITOR
                 [] move -> overlap -> MONITOR)
          []move -> overlap -> MONITOR

RADAR = startup ->
                (RADARWORKLOOP
                [] move -> RADAR)

RADARWORKLOOP = start_min_wk -> send_signal -> tock -> tock ->
recieve_signal -> start_process_data -> tock -> tock -> end_process_data ->
send_to_fusion -> end_min_wk -> RADARWORKLOOP
                [] start_min_wk -> send_signal -> move -> RADAR
                [] start_min_wk -> send_signal -> tock -> move -> RADAR
                [] start_min_wk -> send_signal -> tock -> tock -> move ->
RADAR
                [] start_min_wk -> send_signal -> tock -> tock ->
recieve_signal -> move -> RADAR
                [] start_min_wk -> send_signal -> tock -> tock ->
recieve_signal -> start_process_data -> move -> RADAR
                [] start_min_wk -> send_signal -> tock -> tock ->
recieve_signal -> start_process_data -> tock -> move -> RADAR
                [] start_min_wk -> send_signal -> tock -> tock ->
recieve_signal -> start_process_data -> tock -> tock -> move -> RADAR
                [] start_min_wk -> send_signal -> tock -> tock ->
recieve_signal -> start_process_data -> tock -> tock -> tock ->
end_process_data -> move -> RADAR
                [] start_min_wk -> send_signal -> tock -> tock ->
recieve_signal -> start_process_data -> tock -> tock -> tock ->
end_process_data -> send_to_fusion -> end_min_wk -> move -> RADAR


RADARREST = startup ->
                (RADARWORKLOOPREST
```

```
[] move -> RADARNORECONF)
```

```
RADARWORKLOOPREST = start_min_wk -> send_signal -> tock -> tock ->
recieve_signal -> start_process_data -> tock -> tock -> end_process_data ->
send_to_fusion -> end_min_wk -> RADARWORKLOOPREST
                [] start_min_wk -> send_signal -> move -> RADARNORECONF
                [] start_min_wk -> send_signal -> tock -> move ->
RADARNORECONF
                [] start_min_wk -> send_signal -> tock -> tock -> move ->
RADARNORECONF
                [] start_min_wk -> send_signal -> tock -> tock ->
recieve_signal -> move -> RADARNORECONF
                [] start_min_wk -> send_signal -> tock -> tock ->
recieve_signal -> start_process_data -> move -> RADARNORECONF
                [] start_min_wk -> send_signal -> tock -> tock ->
recieve_signal -> start_process_data -> tock -> move -> RADARNORECONF
                [] start_min_wk -> send_signal -> tock -> tock ->
recieve_signal -> start_process_data -> tock -> tock -> move ->
RADARNORECONF
                [] start_min_wk -> send_signal -> tock -> tock ->
recieve_signal -> start_process_data -> tock -> tock -> tock ->
end_process_data -> move -> RADARNORECONF
                [] start_min_wk -> send_signal -> tock -> tock ->
recieve_signal -> start_process_data -> tock -> tock -> tock ->
end_process_data -> send_to_fusion -> end_min_wk -> move -> RADARREST

RADARNORECONF = startup ->
                RADARWORKLOOPNORECONF

RADARWORKLOOPNORECONF = start_min_wk -> send_signal -> tock -> tock ->
recieve_signal -> start_process_data -> tock -> tock -> end_process_data ->
send_to_fusion -> end_min_wk -> RADARWORKLOOPREST

GROUND = startup ->
                GROUNDWORKLOOP

GROUNDWORKLOOP = start_min_wk -> recieve_signal -> start_process_data ->
tock -> tock -> end_process_data -> send_to_fusion -> end_min_wk ->
RADARWORKLOOP
                [] start_min_wk -> tock -> recieve_signal ->
start_process_data -> tock -> tock -> end_process_data -> send_to_fusion ->
end_min_wk -> RADARWORKLOOP
                [] start_min_wk -> tock -> tock -> recieve_signal ->
start_process_data -> tock -> tock -> end_process_data -> send_to_fusion ->
end_min_wk -> RADARWORKLOOP
                [] start_min_wk -> tock -> tock -> tock -> recieve_signal ->
start_process_data -> tock -> tock -> end_process_data -> send_to_fusion ->
end_min_wk -> RADARWORKLOOP
                [] start_min_wk -> tock -> tock -> tock -> tock ->
recieve_signal -> start_process_data -> tock -> tock -> end_process_data ->
send_to_fusion -> end_min_wk -> RADARWORKLOOP
                [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
recieve_signal -> start_process_data -> tock -> tock -> end_process_data ->
send_to_fusion -> end_min_wk -> RADARWORKLOOP
                [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
tock -> send_to_fusion -> end_min_wk -> RADARWORKLOOP
                [] start_min_wk -> recieve_signal -> start_process_data ->
tock -> tock -> end_process_data -> send_to_fusion -> end_min_wk -> move ->
GROUND
                [] start_min_wk -> tock -> recieve_signal ->
start_process_data -> tock -> tock -> end_process_data -> send_to_fusion ->
end_min_wk -> move -> GROUND
                [] start_min_wk -> tock -> tock -> recieve_signal ->
start_process_data -> tock -> tock -> end_process_data -> send_to_fusion ->
end_min_wk -> move -> GROUND
                [] start_min_wk -> tock -> tock -> tock -> recieve_signal ->
start_process_data -> tock -> tock -> end_process_data -> send_to_fusion ->
end_min_wk -> move -> GROUND
```

```
                [] start_min_wk -> tock -> tock -> tock -> tock ->
recieve_signal -> start_process_data -> tock -> tock -> end_process_data ->
send_to_fusion -> end_min_wk -> move -> GROUND
                [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
recieve_signal -> start_process_data -> tock -> tock -> end_process_data ->
send_to_fusion -> end_min_wk -> move -> GROUND
                [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
tock -> send_to_fusion -> end_min_wk -> move -> GROUND

FUSION = startup ->
                FUSIONWORKLOOP
```

```
FUSIONWORKLOOP = start_min_wk -> RECONFIGFUSIONWORKLOOP
               [] start_min_wk -> tock -> RECONFIGFUSIONWORKLOOP
               [] start_min_wk -> tock -> move -> FUSION
               [] start_min_wk -> tock -> tock -> RECONFIGFUSIONWORKLOOP
               [] start_min_wk -> tock -> tock -> move -> FUSION
               [] start_min_wk -> tock -> tock -> tock ->
RECONFIGFUSIONWORKLOOP
               [] start_min_wk -> tock -> tock -> tock -> move -> FUSION
               [] start_min_wk -> tock -> tock -> tock -> tock ->
RECONFIGFUSIONWORKLOOP
               [] start_min_wk -> tock -> tock -> tock -> tock -> move ->
FUSION
               [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
RECONFIGFUSIONWORKLOOP
               [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
move -> FUSION
               [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
tock -> RECONFIGFUSIONWORKLOOP
               [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
tock -> move -> FUSION
               [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
tock -> output_decision -> end_min_wk -> FUSIONWORKLOOP

RECONFIGFUSIONWORKLOOP = recieve_signal -> start_process_data -> tock ->
tock -> end_process_data -> start_fuse_data -> tock -> end_fuse_data ->
output_decision -> end_min_wk -> FUSIONWORKLOOP
               [] move -> FUSION
               [] recieve_signal -> move -> FUSION
               [] recieve_signal -> start_process_data -> move -> FUSION
               [] recieve_signal -> start_process_data -> tock -> move ->
FUSION
               [] recieve_signal -> start_process_data -> tock -> tock ->
move -> FUSION
               [] recieve_signal -> start_process_data -> tock -> tock ->
end_process_data -> move -> FUSION
               [] recieve_signal -> start_process_data -> tock -> tock ->
end_process_data -> start_fuse_data -> move -> FUSION
               [] recieve_signal -> start_process_data -> tock -> tock ->
end_process_data -> start_fuse_data -> tock -> move -> FUSION
               [] recieve_signal -> start_process_data -> tock -> tock ->
end_process_data -> start_fuse_data -> tock -> end_fuse_data -> move ->
FUSION
               [] recieve_signal -> start_process_data -> tock -> tock ->
end_process_data -> start_fuse_data -> tock -> end_fuse_data ->
output_decision -> end_min_wk -> move -> FUSION

FUSIONREST = startup ->
               FUSIONWORKLOOPREST

FUSIONWORKLOOPREST = start_min_wk -> RECONFIGFUSIONWORKLOOPREST
               [] start_min_wk -> tock -> RECONFIGFUSIONWORKLOOPREST
               [] start_min_wk -> tock -> move -> FUSIONNORECONF
               [] start_min_wk -> tock -> tock -> RECONFIGFUSIONWORKLOOPREST
               [] start_min_wk -> tock -> tock -> move -> FUSIONNORECONF
               [] start_min_wk -> tock -> tock -> tock ->
RECONFIGFUSIONWORKLOOPREST
               [] start_min_wk -> tock -> tock -> tock -> move ->
FUSIONNORECONF
               [] start_min_wk -> tock -> tock -> tock -> tock ->
RECONFIGFUSIONWORKLOOPREST
               [] start_min_wk -> tock -> tock -> tock -> tock -> move ->
FUSIONNORECONF
               [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
RECONFIGFUSIONWORKLOOPREST
               [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
move -> FUSIONNORECONF
               [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
tock -> RECONFIGFUSIONWORKLOOPREST
```

```
                [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
tock -> move -> FUSIONNORECONF
                [] start_min_wk -> tock -> tock -> tock -> tock -> tock ->
tock -> output_decision -> end_min_wk -> FUSIONWORKLOOPREST
```

```
RECONFIGFUSIONWORKLOOPREST = recieve_signal -> start_process_data -> tock ->
tock -> end_process_data -> start_fuse_data -> tock -> end_fuse_data ->
output_decision -> end_min_wk -> FUSIONWORKLOOPREST
                [] move -> FUSIONNORECONF
                [] recieve_signal -> move -> FUSIONNORECONF
                [] recieve_signal -> start_process_data -> move ->
FUSIONNORECONF
                [] recieve_signal -> start_process_data -> tock -> move ->
FUSIONNORECONF
                [] recieve_signal -> start_process_data -> tock -> tock ->
move -> FUSIONNORECONF
                [] recieve_signal -> start_process_data -> tock -> tock ->
end_process_data -> move -> FUSIONNORECONF
                [] recieve_signal -> start_process_data -> tock -> tock ->
end_process_data -> start_fuse_data -> move -> FUSIONNORECONF
                [] recieve_signal -> start_process_data -> tock -> tock ->
end_process_data -> start_fuse_data -> tock -> move -> FUSIONNORECONF
                [] recieve_signal -> start_process_data -> tock -> tock ->
end_process_data -> start_fuse_data -> tock -> end_fuse_data -> move ->
FUSIONNORECONF
                [] recieve_signal -> start_process_data -> tock -> tock ->
end_process_data -> start_fuse_data -> tock -> end_fuse_data ->
output_decision -> end_min_wk -> move -> FUSIONNORECONF

FUSIONNORECONF = startup ->recieve_signal -> start_process_data -> tock ->
tock -> end_process_data -> start_fuse_data -> tock -> end_fuse_data ->
output_decision -> end_min_wk -> FUSIONWORKLOOPREST


SYSTEMRADAR =
(MONITOR|[{startup,move,start_min_wk,end_min_wk}]|(RADAR\{send_to_fusion,
send_signal, recieve_signal, start_process_data,
end_process_data}))\{end_min_wk,startup,start_min_wk,move}|[{tock}]|TOCKS

SYSTEMRADARREST =
(MONITOR|[{startup,move,start_min_wk,end_min_wk}]|(RADARREST\{send_to_fusion
, send_signal, recieve_signal, start_process_data,
end_process_data}))\{end_min_wk,startup,start_min_wk,move}|[{tock}]|TOCKS

SYSTEMGROUND =
(MONITOR|[{startup,move,start_min_wk,end_min_wk}]|(GROUND\{send_to_fusion,
send_signal, recieve_signal, start_process_data,
end_process_data}))\{end_min_wk, startup,start_min_wk,move}|[{tock}]|TOCKS

SYSTEMFUSION =
(MONITOR|[{startup,move,start_min_wk,end_min_wk}]|(FUSION\{output_decision,
start_fuse_data, end_fuse_data, send_signal, recieve_signal,
start_process_data, end_process_data}))\{end_min_wk,
startup,start_min_wk,move}|[{tock}]|TOCKS

SYSTEMFUSIONREST =
(MONITOR|[{startup,move,start_min_wk,end_min_wk}]|(FUSIONREST\{output_decisi
on, start_fuse_data, end_fuse_data, send_signal, recieve_signal,
start_process_data, end_process_data}))\{end_min_wk,
startup,start_min_wk,move}|[{tock}]|TOCKS

THRASHTIMED(<>, maxt, maxo) = overlap -> THRASHTIMED(<>^<O>,maxt,maxo)
                  [] tock -> THRASHTIMED(<>^<T>,maxt,maxo)

THRASHTIMED(x, maxt, maxo) = if (numo(x) == maxo) then
                     thrashbang -> STOP
                   else
                     overlap -> THRASHTIMED(x^<O>,maxt,maxo)
                     [] tock -> if (numt(x) == maxt) then
                          THRASHTIMED(slidewindow(x)^<T>,maxt,maxo)
                        else
                          THRASHTIMED(x^<T>,maxt,maxo)
```

```
numt(x) = length(<y | y <- x, y==T>)

numo(x) = length(<y | y <- x, y==O>)

slidewindow(x) = if (head(x) == T) then
                   tail(x)
                 else
                   slidewindow(tail(x))
```

```
TESTRADAR = (SYSTEMRADAR|[{overlap,
tock}]|THRASHTIMED(<>,10,2))\{tock,overlap}

TESTRADARREST = (SYSTEMRADARREST|[{overlap,
tock}]|THRASHTIMED(<>,10,2))\{tock,overlap}

TESTGROUND = (SYSTEMGROUND|[{overlap,
tock}]|THRASHTIMED(<>,10,2))\{tock,overlap}

TESTFUSION = (SYSTEMFUSION|[{overlap,
tock}]|THRASHTIMED(<>,20,2))\{tock,overlap}

TESTFUSIONREST = (SYSTEMFUSIONREST|[{overlap,
tock}]|THRASHTIMED(<>,20,2))\{tock,overlap}

assert STOP [T= TESTRADAR
assert STOP [T= TESTRADARREST
assert STOP [T= TESTGROUND
assert STOP [T= TESTFUSION
assert STOP [T= TESTFUSIONREST
```

2. *Java RMI Case Study Code*

2.1 Start.java

```
//TEST JAVA FILE
//Role: Locates Controller and Adds Rules for proc1
//      also enables rule checking

import jcode.process.*;
import jcode.controller.Controller;
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;


public class Start
{
  public static void main(String[] args)
  {
    try
    {
      if (args.length != 1)
      {
        System.out.println("Wrong Arguments - Try Again");
        return;
      }
      System.out.println("Starting Test");
      Controller cont = (Controller)
Naming.lookup("rmi://localhost:1099/Controller".concat(args[0]));
      System.out.println("controller reference obtained");
      ReconfigProcess fdm1 = cont.createProcess("FusionDM", "FusionDM");
      try
      {
              Thread.sleep(2000);
      }catch (Exception e)
      {}
      ReconfigProcess p1 = cont.createProcess("GroundSensor", "g1");
      ReconfigProcess r1 = cont.createProcess("RadarSensor", "r1");
      //cont.EnableRules();
      //Object[] x = new Object[5];
      //x[0] = "ConfOver";
      //x[1] = "proc1";
      //x[2] = new Integer(2);
```

```
      //x[3] = new Integer(170);
      //x[4] = new Integer(51);
      //cont.AddRule(x);

    }
    catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }
}
```

2.2 Reconfigure.java

```
//TEST JAVA FILE
//Role: Locates Controller and Adds Rules for proc1
//      also enables rule checking

import jcode.process.*;
import jcode.controller.Controller;
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;


public class Reconfigure
{
  public static void main(String[] args)
  {
    try
    {
      if (args.length != 1)
      {
        System.out.println("Wrong Arguments - Try Again");
        return;
      }
      System.out.println("Starting Test");
      Controller cont = (Controller)
Naming.lookup("rmi://localhost:1099/Controller".concat(args[0]));
      System.out.println("controller reference obtained");
      int i = 0;
      while(true)
      {
        i = i+1;
        System.out.println(i);
        cont.MoveProcDelAfter("FusionDM", "FusionDM", "localhost");
        try
        {
                    Thread.sleep(3050);
        }catch (Exception e)
        {}
        if (i > 30)
        {
          break;
        }
      }
      //ReconfigProcess p1 = cont.createProcess("GroundSensor", "g1");
      //ReconfigProcess r1 = cont.createProcess("RadarSensor", "r1");
      //cont.EnableRules();
      //Object[] x = new Object[5];
      //x[0] = "ConfOver";
      //x[1] = "proc1";
      //x[2] = new Integer(2);
      //x[3] = new Integer(170);
      //x[4] = new Integer(51);
      //cont.AddRule(x);
```

```
    }
    catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }
}
```

2.3 StartCont.java

```
//TEST JAVA FILE
//Role: Locates Controller and Adds Rules for proc1
//      also enables rule checking

import jcode.process.*;
import jcode.controller.Controller;
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;


public class StartCont
{
  public static void main(String[] args)
  {
    try
    {
      if (args.length != 1)
      {
        System.out.println("Wrong Arguments - Try Again");
        return;
      }
      System.out.println("Starting Test");
      Controller cont = (Controller)
Naming.lookup("rmi://localhost:1099/Controller".concat(args[0]));
      System.out.println("controller reference obtained");

      cont.EnableRules();
      Object[] x = new Object[5];
      x[0] = "ConfOver";
      x[1] = "FusionDM";
      x[2] = new Integer(2);
      x[3] = new Integer(100);
      x[4] = new Integer(50);
      cont.AddRule(x);


      ReconfigProcess fdm1 = cont.createProcess("FusionDM", "FusionDM");
      try
      {
              Thread.sleep(2000);
      }catch (Exception e)
      {}
      ReconfigProcess p1 = cont.createProcess("GroundSensor", "g1");
      ReconfigProcess r1 = cont.createProcess("RadarSensor", "r1");
      //cont.EnableRules();
      //Object[] x = new Object[5];
      //x[0] = "ConfOver";
      //x[1] = "proc1";
      //x[2] = new Integer(2);
      //x[3] = new Integer(170);
      //x[4] = new Integer(51);
      //cont.AddRule(x);

    }
    catch (Exception e)
    {
```

```
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }
}
```

2.4 Controller.java

```
package jcode.controller;
import jcode.process.*;
import java.rmi.*;

public interface Controller extends Remote
{
  public ReconfigProcess createProcess(String Type, String ProcName) throws
RemoteException;
  public void bindProcess(ReconfigProcess a, String newloc, String ProcName)
throws RemoteException;
  public void MoveProcDelFirst(String Type, String RMIname, String newloc)
throws RemoteException;
  public void MoveProcDelAfter(String Type, String RMIname, String newloc)
throws RemoteException;
  public void MoveProcWState(String Type, String RMIname, String newloc)
throws RemoteException;
  public void EnableRules() throws RemoteException;
  public void DisableRules() throws RemoteException;
  public void AddRule(Object[] newrule) throws RemoteException;
  public void RemoveRule(Object[] remrule) throws RemoteException;
}
```

2.5 ControllerImpl.java

```
package jcode.controller;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

import java.rmi.server.UnicastRemoteObject;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.util.LinkedList;
import java.util.Date;
import jcode.process.*;
import jcode.factory.*;

class ControllerImpl extends UnicastRemoteObject implements Controller
{

  int rules = 0;
  LinkedList RuleStore = new LinkedList();

  //History stored as Longs in the LinkedList
  //This does not allow for control of multiple
  //processes - though alterations to the code
  //to provide a LinkedList for each process
  //would allow this.  This code was not needed
  //for the demonstrator.
  LinkedList History = new LinkedList();

  //Constructor
  public ControllerImpl() throws RemoteException
  {
    super();
  }

  //Main - program creates its own RMI registry
  public static void main(String[] args)
  {
```

```
    try
    {
      System.out.println("Controller started");
      String a = new String("Controller");
      String cn = a.concat(args[0]);
       System.out.println("Controller will bound to localhost as
".concat(cn));
      LocateRegistry.getRegistry().bind(cn, new ControllerImpl());
      System.out.println("Controller bound to localhost as ".concat(cn));
    } catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }

  public void EnableRules() throws RemoteException
  {
    this.rules=1;
  }

  public void DisableRules() throws RemoteException
  {
    this.rules=0;
  }

  private int ReconfigYN(String RMIname)
  {
    if (this.rules == 0)
    {
      //rules not applicable thus reconfiguration fine
      System.out.println("Rules Not Applicable - Reconfiguration Fine");
      return 1;
    }
    else
    {
      //need to check rules
      if (RuleStore.size() == 0)
      {
        System.out.println("No Rules Exist");
        return 1;
      }
      for (int i = 0;i<RuleStore.size(); i++)
      {
        Object[] Rule = (Object[])RuleStore.get(i);
        //rules with knock on effects like collocated should be added twice
        //to simplify checks
        if (((String)Rule[1]).compareTo(RMIname) == 0)
        {
          System.out.println("FOUND A RULE THAT APPLIES");
          //only one type of rule presently available
          if (((String)Rule[0]).compareTo("ConfOver") == 0)
          {
            int x = ((Integer)Rule[2]).intValue();
            int y = ((Integer)Rule[3]).intValue();
            int z = ((Integer)Rule[4]).intValue();

            Date date = new Date();
            long currentTime = date.getTime();
            long lastReconfig = 0;

            if (History.size() == 0)
            {
              System.out.println("History Empty - Not an overlap");
              History.add(new Long(currentTime));
              return 1;
            }
            else
            {
```

```
            lastReconfig = ((Long)History.getLast()).longValue();
            if ((currentTime - lastReconfig) > ((long)z)*1000)
            {
              System.out.println("Not an overlap");
              History.add(new Long(currentTime));
              return 1;
            }
            else
            {
              int count = 0;
              long stInterval = currentTime - (y*1000);
              for (int a=0; a<History.size(); a++)
              {
                if (((Long)History.get(a)).longValue() > stInterval)
                {
                  System.out.println("ADD ONE");
                  count++;
                }
              }
              if (count < x)
              {
                System.out.println("Overlap Occured");
                History.add(new Long(currentTime));
                return 1;
              }
            }
          }
          //Need overlaps in y interval
          System.out.println("Reconfiguration Denied");
          return 0;
        }
      }
    }
    return 1;
  }
}

public void AddRule(Object[] newrule) throws RemoteException
{
  //for live system checks for rule well formedness would be required
  this.RuleStore.add(newrule);
}

public void RemoveRule(Object[] remrule) throws RemoteException
{
  if (this.RuleStore.contains(remrule))
  {
    this.RuleStore.remove(remrule);
  }
  else
  {
    System.out.println("Rule Does not Exist");
  }
}

  public ReconfigProcess createProcess(String Type, String ProcName) throws
RemoteException
  {
    try
    {
      System.out.println("Trying to Create Process");
      ProcessFactory fact = (ProcessFactory)
Naming.lookup("rmi://localhost:1099/ProcessFactory");
      ReconfigProcess a = (ReconfigProcess) fact.createProcess(Type,
ProcName);
      System.out.println("New Process Created");
      this.bindProcess(a, "localhost", ProcName);
      return a;
    }catch(Exception e)
```

```
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
      throw new RemoteException("Process Cannot Be Created");
    }
  }

  //Function to enable del after functions - multiple binds not possible in
RMIRegistry
  private ReconfigProcess createProcNoBind(String Type, String ProcName)
throws RemoteException
  {
    try
    {
      System.out.println("Trying to Create Process");
      ProcessFactory fact = (ProcessFactory)
Naming.lookup("rmi://localhost:1099/ProcessFactory");
      ReconfigProcess a = (ReconfigProcess)fact.createProcess(Type,
ProcName);
      System.out.println("New Process Created");
      //this.bindProcess(a, "localhost");
      return a;
    }catch(Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
      throw new RemoteException("Process Cannot Be Created");
    }
  }

  public void bindProcess(ReconfigProcess a, String loc, String ProcName)
throws RemoteException
  {
    try
    {
      String cn = ProcName;
      LocateRegistry.getRegistry(loc).bind(cn, a);
      System.out.println("ReconfigProcess bound to".concat(loc.concat(" as
".concat(cn))));
    }catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }

  public void MoveProcDelFirst(String Type, String RMIname, String newloc)
throws RemoteException
  {
    //Check Reconfiguration can occur
    if (this.ReconfigYN(RMIname) == 1)
    {
      try
      {
        ReconfigProcess p = (ReconfigProcess)
Naming.lookup("rmi://localhost:1099/".concat(RMIname));
        System.out.println("ReconfigProcess reference obtained");
        p.delProcess();
        System.out.println("Process Deleted");
        ReconfigProcess p1 = this.createProcess(Type, RMIname);
        System.out.println("New Process Created");
      }catch (Exception e)
      {
        System.out.println("Exception caught: " + e);
        e.printStackTrace();
      }
    }
    else
    {
```

```
      System.out.println("Reconfiguration Denied");
    }
  }

  public void MoveProcDelAfter(String Type, String RMIname, String newloc)
throws RemoteException
  {
    if (this.ReconfigYN(RMIname) == 1)
    {
      try
      {
        ReconfigProcess p1 = this.createProcNoBind(Type, RMIname);
        System.out.println("New Process Created");
        ReconfigProcess p = (ReconfigProcess)
Naming.lookup("rmi://localhost:1099/".concat(RMIname));
        System.out.println("ReconfigProcess reference obtained");
        p.delProcess();
        System.out.println("Process Deleted");
        //bind must take place last as RMIRegistry does not allow multiple
binds to same name.
        this.bindProcess(p1, "localhost", RMIname);
      }catch (Exception e)
      {
        System.out.println("Exception caught: " + e);
        e.printStackTrace();
      }
    }
    else
    {
      System.out.println("Reconfiguration Denied");
    }
  }

  public void MoveProcWState(String Type, String RMIname, String newloc)
throws RemoteException
  {
    if (this.ReconfigYN(RMIname) == 1)
    {
      try
      {
        ReconfigProcess p1 = this.createProcNoBind(Type, RMIname);
        System.out.println("New Process Created");
        ReconfigProcess p = (ReconfigProcess)
Naming.lookup("rmi://localhost:1099/".concat(RMIname));
        System.out.println("ReconfigProcess reference obtained");
        p1.setData(p.getData());
        System.out.println("Process State Transfered");
        p.delProcess();
        System.out.println("Process Deleted");
        //bind must take place last as RMIRegistry does not allow multiple
binds to same name.
        this.bindProcess(p1, "localhost", RMIname);
      }catch (Exception e)
      {
        System.out.println("Exception caught: " + e);
        e.printStackTrace();
      }
    }
    else
    {
      System.out.println("Reconfiguration Denied");
    }
  }

}
```

2.6 ProcessFactory.java

```
package jcode.factory;
import jcode.process.*;
import java.rmi.*;

public interface ProcessFactory extends Remote
{
  public ReconfigProcess createProcess(String Type, String ProcName)
throws RemoteException;
  public void setProcess(ReconfigProcess a) throws RemoteException;
}
```

2.7 ProcessFactoryImpl.java

```
package jcode.factory;
//import java.rmi.*;
//import java.rmi.server.*;
//import java.rmi.registry.*;

import java.rmi.server.UnicastRemoteObject;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import jcode.process.*;

//public class AgentFactoryImpl extends UnicastRemoteObject implements
AgentFactory
class ProcessFactoryImpl extends UnicastRemoteObject implements
ProcessFactory
{
  int data = 0;

  //Constructor
  public ProcessFactoryImpl() throws RemoteException
  {
    super();
  }

  //Main - program creates its own RMI registry
  public static void main(String[] argv)
  {
    try
    {
      System.out.println("Process factory started");
      //LocateRegistry.createRegistry(1099);
      //System.out.println("Registry started");
      //registry must be started
      LocateRegistry.getRegistry().bind("ProcessFactory",new
ProcessFactoryImpl());
      System.out.println("Process factory bound");
    } catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }

  public ReconfigProcess createProcess(String Type, String ProcName) throws
RemoteException
  {
    ReconfigProcess a;
    System.out.println("Type: "+Type);
    if (Type.equals("GroundSensor"))
    {
       a = (ReconfigProcess)new GroundSensorImpl(ProcName);
    }
    else if (Type.equals("RadarSensor"))
    {
       a = (ReconfigProcess)new RadarSensorImpl(ProcName);
    }
```

```
    else
    {
        a = (ReconfigProcess)new FusionDMImpl(ProcName);
    }
        return (ReconfigProcess)java.rmi.server.RemoteObject.toStub(a) ;
  }

  public void setProcess(ReconfigProcess a) throws RemoteException
  {
    this.data += a.getData();
  }
}
```

2.8 ReconfigProcess.java

```
package jcode.process;
import java.rmi.*;
import jcode.process.*;
import jcode.factory.*;


public interface ReconfigProcess extends Remote
{
  public void delProcess() throws RemoteException;
  public void compute() throws RemoteException;
  public int getData() throws RemoteException;
  public void setData(int data) throws RemoteException;
  public void setGROUNDData(int data) throws RemoteException;
  public void setRADARata(int data) throws RemoteException;
  public int getGROUNDData() throws RemoteException;
  public int getRADARData() throws RemoteException;
}
```

2.9 FusionDMImpl.java

```
package jcode.process;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

import java.rmi.server.UnicastRemoteObject;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import jcode.process.*;
import jcode.factory.*;


public class FusionDMImpl extends UnicastRemoteObject implements
ReconfigProcess
{
  String bindname;
  int data;
  int RDdata = -1;
  int GRdata = -1;
  MainFusionDMThread mt = new MainFusionDMThread();

  //Constructor
  public FusionDMImpl(String bindname) throws RemoteException
  {
    super();
    this.bindname = "FusionDM";
    System.out.println("Fusion DM Initialising");
    mt.setData(this.bindname);
    mt.start();
  }

  //No Main As Process is not meant to run from command line
  public void delProcess() throws RemoteException
```

```
  {
    //No need to call a deconstructor as Java RMI uses a Distributed Garbage
Collection
    //system and thus will remove stale processes once no reference exists
for them.
    try {
      String cn = this.bindname;
      System.out.println("Radar Sensor Uninitialising");
      LocateRegistry.getRegistry().unbind(cn);
      mt.stop();
      System.out.println("Radar Sensor no longer bound to localhost as
".concat(cn));
    }catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }

  public void compute() throws RemoteException
  {
    data += 10;
  }

  public int getData() throws RemoteException
  {
    return data;
  }

  public void setData(int data) throws RemoteException
  {
    this.data = data;
  }

  public void setGROUNDData(int data) throws RemoteException
  {
    this.GRdata = data;
  }

  public void setRADARata(int data) throws RemoteException
  {
    this.RDdata = data;
  }

  public int getGROUNDData() throws RemoteException
  {
    int GRdata1 = this.GRdata;
    //reset GRdata
    this.GRdata = -1;
    return GRdata1;
  }

  public int getRADARData() throws RemoteException
  {
    int RDdata1 = this.RDdata;
    //reset RDdata
    this.RDdata = -1;
    return RDdata;
  }
}
```

2.10 MainFusionDMThread.java

```
package jcode.process;
import java.util.Random;
import javax.rmi.*;
import java.rmi.*;
import java.rmi.server.*;
```

```
import java.net.*;
import java.util.Calendar;
import java.text.SimpleDateFormat;


public class MainFusionDMThread extends Thread
{
  public String link;

  public void setData(String parproc)
  {
    this.link = parproc;
  }

  public void run()
  {
        //Assumed only a single DM Named FusionDM for other processes
simplicity
        Random generator = new Random();
        Calendar cal = Calendar.getInstance();
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        System.out.println("FusionDM -- Start
Time:"+sdf.format(cal.getTime()));

        try
        {
                Thread.sleep(1000);
        }catch (Exception e)
        {}

        while(true)
        {
                try{
                        ReconfigProcess mfc =
(ReconfigProcess)Naming.lookup("rmi://localhost:1099/FusionDM");
                        System.out.println("FusionDM -- own reference
obtained");
                        int output = 0;
                        int i = 0;
                        for(i = 0; i<6; i++)
                        {
                                System.out.println("FusionDM -- Poll for
Input");
                                try
                                {
                                        int grdata = mfc.getGROUNDData();
                                        int rddata = mfc.getRADARData();
                                        if (rddata != -1 || grdata != -1)
                                        {
                                                System.out.println("FusionDM --
got new data");
                                                System.out.println("FusionDM --
start process data");
                                                try
                                                {
                                                        Thread.sleep(6000);
                                                }catch (Exception e)
                                                {}
                                                System.out.println("FusionDM --
end process data");
                                                System.out.println("FusionDM --
start fuse data");
                                                try
                                                {
                                                        Thread.sleep(3000);
                                                }catch (Exception e)
                                                {}
                                                System.out.println("FusionDM --
end fuse data");
```

```
                                        System.out.println("FusionDM --
output Desision based on NON stale data");
                                        Calendar cal1 =
Calendar.getInstance();
                                        SimpleDateFormat sdf1 = new
SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
                                        System.out.println("FusionDM --
Output Time:"+sdf1.format(cal1.getTime()));
                                        output = 1;
                                        i=10;
                                        break;
                                }
                                else
                                {
                                        System.out.println("FusionDM --
no new data");
                                        //wait for new data - hard
coded time
                                        try
                                        {
                                                Thread.sleep(3000);
                                        }catch (Exception e)
                                        {}
                                }
                        }catch (Exception e)
                        {
                                System.out.println("Exception caught: "
+ e);
                                e.printStackTrace();
                                break;
                        }
                }
                if (output == 0)
                {
                System.out.println("FusionDM -- output Desision based
on stale data");
                Calendar cal2 = Calendar.getInstance();
                SimpleDateFormat sdf2 = new SimpleDateFormat("yyyy-MM-
dd HH:mm:ss");
                System.out.println("FusionDM -- Output
Time:"+sdf2.format(cal2.getTime()));
                }
        }catch (Exception e)
        {
                System.out.println("Exception caught: " + e);
                e.printStackTrace();
                break;
        }
    }
  }
}
```

2.11 GroundSensorImpl.java

```
package jcode.process;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

import java.rmi.server.UnicastRemoteObject;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import jcode.process.*;
import jcode.factory.*;


public class GroundSensorImpl extends UnicastRemoteObject implements
ReconfigProcess
```

```
{
  String bindname;
  int data;
  MainGroundThread mt = new MainGroundThread();

  //Constructor
  public GroundSensorImpl(String bindname) throws RemoteException
  {
    super();
    this.bindname = bindname;
    System.out.println("Ground Sensor Initialising");
    mt.setData(bindname);
    mt.start();
  }

  //No Main As Process is not meant to run from command line
  public void delProcess() throws RemoteException
  {
    //No need to call a deconstructor as Java RMI uses a Distributed Garbage
Collection
    //system and thus will remove stale processes once no reference exists
for them.
    try {
      String cn = this.bindname;
      System.out.println("Ground Sensor Uninitialising\r\n");
      LocateRegistry.getRegistry().unbind(cn);
      mt.stop();
      System.out.println("Ground Sensor no longer bound to localhost as
".concat(cn));
    }catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }

  public void compute() throws RemoteException
  {
    data += 10;
  }

  public int getData() throws RemoteException
  {
    return data;
  }

  public void setData(int data) throws RemoteException
  {
    data = data;
  }

  public void setGROUNDData(int data) throws RemoteException
  {
    data = data;
  }

  public void setRADARata(int data) throws RemoteException
  {
    data = data;
  }

  public int getGROUNDData() throws RemoteException
  {
    return data;
  }

  public int getRADARData() throws RemoteException
  {
    return data;
```

```
  }

}
```

2.12 MainGroundSensorThread.java

```
package jcode.process;
import java.util.Random;
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
import jcode.controller.Controller;

public class MainGroundThread extends Thread
{
  public String link;

  public void setData(String parproc)
  {
    this.link = parproc;
  }

  public void run()
  {
        Random generator = new Random();

        while(true)
        {
                int sound = 0;
                 int i=0;
                for(i = 0; i<5; i++)
                {
                        System.out.println("GROUND -- Poll for Sound");
                        //wait for sound - hard coded time
                        try
                        {
                                Thread.sleep(3000);
                        }catch (Exception e)
                        {}

                        //Generate random number
                        int Data = generator.nextInt();

                        //If even sound detected
                        if(Data % 2 == 0)
                        {
                                sound = 1;
                                break;
                        }
                }

                if (sound == 1)
                {
                        System.out.println("GROUND -- Ground Sensor Got Sound
- Process Data Start");

                        //wait for processing - hard coded time
                        try
                        {
                                Thread.sleep(6000);
                        }catch (Exception e)
                        {}

                        System.out.println("GROUND -- Ground Sensor Process
Data End");

                        System.out.println("GROUND -- SEND DATA TO FUSION");
                        try{
```

```
                        ReconfigProcess mfc =
(ReconfigProcess)Naming.lookup("rmi://localhost:1099/FusionDM");
                        System.out.println("GROUND -- Fusion DM
reference obtained");
                        mfc.setGROUNDData(1);
                        System.out.println("GROUND -- SENT DATA TO
FUSION");
                    }catch (Exception e)
                    {}


                    sound = 0;
                }
                else
                {
                    System.out.println("GROUND -- SEND DATA TO FUSION
SAYING NO SOUND");
                    try{
                        ReconfigProcess mfc =
(ReconfigProcess)Naming.lookup("rmi://localhost:1099/FusionDM");
                        System.out.println("GROUND -- Fusion DM
reference obtained");
                        mfc.setGROUNDData(0);
                        System.out.println("GROUND -- SENT DATA TO
FUSION");
                    }catch (Exception e)
                    {}
                }

                int Data1 = generator.nextInt();
                //if fault
                if(Data1 % 2 == 0)
                {
                    System.out.println("GROUND -- DETECTED FAULT -
RECONFIGURING");
                    try{
                        Controller cont = (Controller)
Naming.lookup("rmi://localhost:1099/Controllercontroller");
                        System.out.println("GROUND -- controller
reference obtained");
                        cont.MoveProcDelAfter("GroundSensor",
this.link,"localhost");
                    }
                    catch (Exception e)
                    {
                        System.out.println("Exception caught: " + e);
                        e.printStackTrace();
                    }
                }
        }
    }
}
```

2.13 RadarSensorImpl.java

```
package jcode.process;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

import java.rmi.server.UnicastRemoteObject;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import jcode.process.*;
import jcode.factory.*;
```

```
public class RadarSensorImpl extends UnicastRemoteObject implements
ReconfigProcess
{
  String bindname;
  int data;
  MainRadarThread mt = new MainRadarThread();

  //Constructor
  public RadarSensorImpl(String bindname) throws RemoteException
  {
    super();
    this.bindname = bindname;
    System.out.println("Radar Sensor Initialising");
    mt.start();
  }

  //No Main As Process is not meant to run from command line
  public void delProcess() throws RemoteException
  {
    //No need to call a deconstructor as Java RMI uses a Distributed Garbage
Collection
    //system and thus will remove stale processes once no reference exists
for them.
    try {
      String cn = this.bindname;
      System.out.println("Radar Sensor Uninitialising");
      LocateRegistry.getRegistry().unbind(cn);
      mt.stop();
      System.out.println("Radar Sensor no longer bound to localhost as
".concat(cn));
    }catch (Exception e)
    {
      System.out.println("Exception caught: " + e);
      e.printStackTrace();
    }
  }

  public void compute() throws RemoteException
  {
    data += 10;
  }

  public int getData() throws RemoteException
  {
    return data;
  }

  public void setData(int data) throws RemoteException
  {
    this.data = data;
  }

  public void setGROUNDData(int data) throws RemoteException
  {
    data = data;
  }

  public void setRADARata(int data) throws RemoteException
  {
    data = data;
  }

  public int getGROUNDData() throws RemoteException
  {
    return data;
  }

  public int getRADARData() throws RemoteException
  {
```

```
    return data;
  }

}
```

2.14 MainRadarThread.java

```
package jcode.process;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class MainRadarThread extends Thread
{

  public void run()
  {
        while(true)
        {
                System.out.println("RADAR -- Radar Sent");
                //wait for response - hard coded time
                try
                {
                        Thread.sleep(6000);
                }catch (Exception e)
                {}

                System.out.println("RADAR -- Radar Data Recieved");
                System.out.println("RADAR -- Radar Process Data Start");

                //wait for processing - hard coded time
                try
                {
                         Thread.sleep(9000);
                }catch (Exception e)
                {}

                System.out.println("RADAR -- Radar Process Data End");

                System.out.println("RADAR -- SEND DATA TO FUSION");
                System.out.println("GROUND -- SEND DATA TO FUSION");
                try{
                        ReconfigProcess mfc =
(ReconfigProcess)Naming.lookup("rmi://localhost:1099/FusionDM");
                        System.out.println("GROUND -- Fusion DM reference
obtained");
                        mfc.setRADARata(1);
                        System.out.println("GROUND -- SENT DATA TO FUSION");
                }catch (Exception e)
                {}
        }
  }
}
```

3. *Java RMI Case Study Code Outputs*

3.1 Unconstrained Output

```
Process factory started
Process factory bound
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:00
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
Type: GroundSensor
```

```
Ground Sensor Initialising
GROUND -- Poll for Sound
Type: RadarSensor
Radar Sensor Initialising
RADAR -- Radar Sent
FusionDM -- Poll for Input
FusionDM -- no new data
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:04
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
GROUND -- Poll for Sound
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:07
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
GROUND -- Poll for Sound
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:11
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
GROUND -- Poll for Sound
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:14
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
GROUND -- Poll for Sound
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:17
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
GROUND -- SEND DATA TO FUSION SAYING NO SOUND
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:20
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:23
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
RADAR -- Radar Data Recieved
```

```
RADAR -- Radar Process Data Start
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:26
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:29
GROUND -- Poll for Sound
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Got Sound - Process Data Start
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:32
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:35
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- DETECTED FAULT - RECONFIGURING
GROUND -- controller reference obtained
Type: GroundSensor
Ground Sensor Initialising
GROUND -- Poll for Sound
Ground Sensor Uninitialising

Ground Sensor no longer bound to localhost as g1
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:38
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Poll for Sound
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:41
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
```

```
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Poll for Sound
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:44
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Poll for Sound
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:47
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Poll for Sound
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:50
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Got Sound - Process Data Start
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:53
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:31:57
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:32:00
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Got Sound - Process Data Start
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:32:03
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
```

```
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:32:06
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- DETECTED FAULT - RECONFIGURING
GROUND -- controller reference obtained
Type: GroundSensor
Ground Sensor Initialising
GROUND -- Poll for Sound
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
Ground Sensor Uninitialising

Ground Sensor no longer bound to localhost as g1
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:32:09
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Got Sound - Process Data Start
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:32:12
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:32:15
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:32:18
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Got Sound - Process Data Start
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:32:21
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
RADAR -- Radar Data Recieved
```

```
RADAR -- Radar Process Data Start
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:32:24
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- DETECTED FAULT - RECONFIGURING
GROUND -- controller reference obtained
Type: GroundSensor
Ground Sensor Initialising
GROUND -- Poll for Sound
Ground Sensor Uninitialising

Ground Sensor no longer bound to localhost as g1
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:32:27
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Poll for Sound
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:32:30
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
GROUND -- Poll for Sound
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:32:33
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Got Sound - Process Data Start
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:32:36
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Poll for Sound
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
```

```
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
GROUND -- Poll for Sound
FusionDM -- end process data
FusionDM -- start fuse data
GROUND -- Poll for Sound
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:32:52
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- Poll for Input
FusionDM -- no new data
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- end process data
FusionDM -- start fuse data
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:33:10
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Poll for Sound
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- end process data
FusionDM -- start fuse data
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:33:19
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- DETECTED FAULT - RECONFIGURING
GROUND -- controller reference obtained
Type: GroundSensor
Ground Sensor Initialising
GROUND -- Poll for Sound
Ground Sensor Uninitialising

Ground Sensor no longer bound to localhost as g1
FusionDM -- Poll for Input
```

```
FusionDM -- got new data
FusionDM -- start process data
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- end process data
FusionDM -- start fuse data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:33:31
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
GROUND -- Poll for Sound
GROUND -- Poll for Sound
FusionDM -- end process data
FusionDM -- start fuse data
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:33:40
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
GROUND -- Poll for Sound
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- end process data
FusionDM -- start fuse data
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:33:55
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- DETECTED FAULT - RECONFIGURING
GROUND -- controller reference obtained
Type: GroundSensor
Ground Sensor Initialising
GROUND -- Poll for Sound
Ground Sensor Uninitialising

Ground Sensor no longer bound to localhost as g1
FusionDM -- Poll for Input
```

```
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Ground Sensor Got Sound - Process Data Start
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
FusionDM -- end process data
FusionDM -- start fuse data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:34:07
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
GROUND -- Poll for Sound
GROUND -- Poll for Sound
FusionDM -- end process data
FusionDM -- start fuse data
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:34:16
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- DETECTED FAULT - RECONFIGURING
GROUND -- controller reference obtained
Type: GroundSensor
Ground Sensor Initialising
GROUND -- Poll for Sound
Ground Sensor Uninitialising

Ground Sensor no longer bound to localhost as g1
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- end process data
FusionDM -- start fuse data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:34:31
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
```

```
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- end process data
FusionDM -- start fuse data
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- DETECTED FAULT - RECONFIGURING
GROUND -- controller reference obtained
Type: GroundSensor
Ground Sensor Initialising
GROUND -- Poll for Sound
Ground Sensor Uninitialising

Ground Sensor no longer bound to localhost as g1
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:34:40
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Poll for Sound
GROUND -- Poll for Sound
FusionDM -- end process data
FusionDM -- start fuse data
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
GROUND -- Poll for Sound
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:34:49
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Poll for Sound
FusionDM -- Poll for Input
FusionDM -- no new data
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
GROUND -- SEND DATA TO FUSION SAYING NO SOUND
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- DETECTED FAULT - RECONFIGURING
GROUND -- controller reference obtained
Type: GroundSensor
Ground Sensor Initialising
GROUND -- Poll for Sound
Ground Sensor Uninitialising

Ground Sensor no longer bound to localhost as g1
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- end process data
FusionDM -- start fuse data
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
```

```
FusionDM -- Output Time:2009-01-19 01:35:04
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Ground Sensor Got Sound - Process Data Start
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
FusionDM -- end process data
FusionDM -- start fuse data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:35:13
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Poll for Sound
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
GROUND -- Poll for Sound
FusionDM -- end process data
FusionDM -- start fuse data
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:35:22
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- DETECTED FAULT - RECONFIGURING
GROUND -- controller reference obtained
Type: GroundSensor
Ground Sensor Initialising
GROUND -- Poll for Sound
Ground Sensor Uninitialising

Ground Sensor no longer bound to localhost as g1
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Poll for Sound
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
GROUND -- Poll for Sound
FusionDM -- end process data
FusionDM -- start fuse data
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:35:37
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
FusionDM -- Poll for Input
FusionDM -- no new data
```

```
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Ground Sensor Got Sound - Process Data Start
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
FusionDM -- end process data
FusionDM -- start fuse data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- DETECTED FAULT - RECONFIGURING
GROUND -- controller reference obtained
Type: GroundSensor
Ground Sensor Initialising
GROUND -- Poll for Sound
Ground Sensor Uninitialising

Ground Sensor no longer bound to localhost as g1
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:35:52
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
GROUND -- Poll for Sound
GROUND -- Poll for Sound
FusionDM -- end process data
FusionDM -- start fuse data
GROUND -- Poll for Sound
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:36:01
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- Poll for Input
FusionDM -- no new data
FusionDM -- Poll for Input
FusionDM -- no new data
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- DETECTED FAULT - RECONFIGURING
GROUND -- controller reference obtained
Type: GroundSensor
Ground Sensor Initialising
GROUND -- Poll for Sound
Ground Sensor Uninitialising

Ground Sensor no longer bound to localhost as g1
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- end process data
```

```
FusionDM -- start fuse data
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:36:19
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Ground Sensor Got Sound - Process Data Start
```

3.2 Constrained Output

```
Process factory started
Process factory bound
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:50:32
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
Type: GroundSensor
Ground Sensor Initialising
GROUND -- Poll for Sound
Type: RadarSensor
Radar Sensor Initialising
RADAR -- Radar Sent
FusionDM -- Poll for Input
FusionDM -- no new data
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:50:37
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:50:40
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Ground Sensor Got Sound - Process Data Start
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
FusionDM -- end process data
FusionDM -- start fuse data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
```

```
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:50:53
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Ground Sensor Got Sound - Process Data Start
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
FusionDM -- end process data
FusionDM -- start fuse data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:51:02
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Poll for Sound
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- end process data
FusionDM -- start fuse data
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:51:11
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- DETECTED FAULT - RECONFIGURING
GROUND -- controller reference obtained
Type: GroundSensor
Ground Sensor Initialising
GROUND -- Poll for Sound
Ground Sensor Uninitialising

Ground Sensor no longer bound to localhost as g1
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Ground Sensor Got Sound - Process Data Start
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
FusionDM -- end process data
FusionDM -- start fuse data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- DETECTED FAULT - RECONFIGURING
GROUND -- controller reference obtained
Type: GroundSensor
Ground Sensor Initialising
GROUND -- Poll for Sound
Ground Sensor Uninitialising
```

```
Ground Sensor no longer bound to localhost as g1
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:51:23
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Ground Sensor Got Sound - Process Data Start
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
FusionDM -- end process data
FusionDM -- start fuse data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- Poll for Sound
Type: FusionDM
Fusion DM Initialising
FusionDM -- Start Time:2009-01-19 01:51:32
Radar Sensor Uninitialising
Radar Sensor no longer bound to localhost as FusionDM
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Poll for Sound
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- Poll for Input
FusionDM -- no new data
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
FusionDM -- Poll for Input
FusionDM -- no new data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- DETECTED FAULT - RECONFIGURING
GROUND -- controller reference obtained
Type: GroundSensor
Ground Sensor Initialising
GROUND -- Poll for Sound
Ground Sensor Uninitialising

Ground Sensor no longer bound to localhost as g1
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Ground Sensor Got Sound - Process Data Start
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
FusionDM -- end process data
FusionDM -- start fuse data
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- DETECTED FAULT - RECONFIGURING
GROUND -- controller reference obtained
Type: GroundSensor
Ground Sensor Initialising
GROUND -- Poll for Sound
Ground Sensor Uninitialising

Ground Sensor no longer bound to localhost as g1
```

```
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:51:54
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
GROUND -- Poll for Sound
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- end process data
FusionDM -- start fuse data
FusionDM -- end fuse data
FusionDM -- output Desision based on NON stale data
FusionDM -- Output Time:2009-01-19 01:52:03
FusionDM -- own reference obtained
FusionDM -- Poll for Input
FusionDM -- no new data
RADAR -- Radar Process Data End
RADAR -- SEND DATA TO FUSION
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
RADAR -- Radar Sent
GROUND -- Ground Sensor Process Data End
GROUND -- SEND DATA TO FUSION
GROUND -- Fusion DM reference obtained
GROUND -- SENT DATA TO FUSION
GROUND -- DETECTED FAULT - RECONFIGURING
GROUND -- controller reference obtained
Type: GroundSensor
Ground Sensor Initialising
GROUND -- Poll for Sound
Ground Sensor Uninitialising

Ground Sensor no longer bound to localhost as g1
FusionDM -- Poll for Input
FusionDM -- got new data
FusionDM -- start process data
GROUND -- Poll for Sound
RADAR -- Radar Data Recieved
RADAR -- Radar Process Data Start
GROUND -- Ground Sensor Got Sound - Process Data Start
FusionDM -- end process data
FusionDM -- start fuse data
```