Research Commons

http://researchcommons.waikato.ac.nz/

**Research Commons at the University of Waikato**

## Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from the thesis.

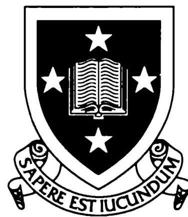
# A Distributed Adaptive Debugger Server

A thesis presented to
The University of Waikato
in fulfilment of the thesis requirement
for the degree of

Doctor of Philosophy

by

**Derek Antony Wong**




The University of Waikato

**Abstract**

*A model and corresponding protocol suitable for debugging distributed embedded systems is developed. Protocol design issues and a formal model of the debugger server are defined with the intent of developing an extensible debug server to aid the development of dynamically retargettable debug clients suitable for application to heterogeneous distributed systems.*

*The thesis considers one of the major problems faced in embedded systems, that of dynamic configuration of system components. The namespace model described solves these issues in a way that provides for the development of dynamic "hot insertion, plug and play" systems. This model is applied throughout the design of the protocol toward the development of debug services.*

*The model is formally developed using the specification language, VDM-SL. Specifications of the model are given implicitly and then derived to an explicit executable form except for target-dependencies. Informal correctness proofs are carried out where necessary to show that the transformation from implicit to explicit form is valid and that the model and protocols themselves are valid. The specifications have been syntax and type checked using the IFAD VDM Toolbox; a CASE tool based on the 1996 ISO VDM-SL standard that provides an environment for the development and testing of VDM-SL specifications.*

## Acknowledgements

Thanks to Mum, Dad and the rest of the family for encouraging me to carry out my work. They have always embraced the value of education and I appreciate the opportunity they have given me through the sacrifices they have made throughout their lives.

A big thank you to my wife, Choi Foong Chen who has put up with my work for such a long time. First as girlfriend and now as wife! I very much appreciate your faith in me and your encouragement.

Close friends and work colleagues also deserve mention for their ideas, encouragement and (sometimes) not so gentle pushing to get the work done. Thanks to them, life was fun, making work considerably more enjoyable. Many thanks to my employers, Scitec (N.Z.) Ltd. and Dialogic (N.Z.) Ltd. for allowing me the time off work to revise this thesis and for providing me with a background of experience that has allowed me to evaluate my work from a different viewpoint.

The University of Waikato's support through the University of Waikato Post-Graduate Scholarship is gratefully acknowledged. Staff of the Computer Science Department also deserve a mention for their help and assistance during my undergraduate years right through to my post-graduate work. Thank you to my supervisory panel for reading drafts and querying any of my nonsense early on. Special thanks go to Dr. Robert Barbour for the time he put into proof reading parts of this thesis at short notice.

The greatest thanks goes to my major supervisor, Mr. Keith Hopper who has been of tremendous support throughout this research. The amount of time he has had for me, being there as a sounding board for my ideas is greatly appreciated. His attention to detail and a "get it right, keep it simple-stupid" attitude has kept me on the right track whenever I was going off the rails! His contribution as supervisor and a friend will always be remembered.

# Table of Contents

**Abstract** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **iii**
**Acknowledgements** . . . . . . . . . . . . . . . . . . . . . . . . **v**
**Table of Contents** . . . . . . . . . . . . . . . . . . . . . . . . **vii**

**Chapters**

**1. Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . **1**
*Origins of this Thesis* . . . . . . . . . . . . . . . . . . . . . . *1*
*Distributed and Embedded Computing* . . . . . . . . . . . . . . . *2*
*The Task of Debugging* . . . . . . . . . . . . . . . . . . . . . . *3*
*The Debugging Environment* . . . . . . . . . . . . . . . . . . . . *4*
*Debugging Instrumentation—Tools of the Trade* . . . . . . . . . . *4*
*Thesis Domain* . . . . . . . . . . . . . . . . . . . . . . . . . . *5*
*A Model and a Protocol* . . . . . . . . . . . . . . . . . . . . . *5*
*Summary* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *6*

**2. Looking for an Interface** . . . . . . . . . . . . . . . . . . . **7**
*What is Debugging?* . . . . . . . . . . . . . . . . . . . . . . . *7*
*The Problems of Debugging* *7*
*Debugging Tools* . . . . . . . . . . . . . . . . . . . . . . . . . *8*
*Debugging Programs on Single Processor Machines* . . . . . . . . *9*
*Traditional Uniprocessor Debugging Architecture* *10*
*Debugging Parallel and Distributed Programs* . . . . . . . . . . *10*
*Distributed Debugging Architecture* *11*
*Embedded Systems and Debugging* *12*
*Defining a Common Interface* . . . . . . . . . . . . . . . . . . . *13*
*A Taxonomy for Classifying Embedded Distributed Debuggers* . . . *13*
*Hardware Based Monitors and Debuggers* *17*
*The Test and Measurement Processor* *18*
*The Topaz Teledebugger* . . . . . . . . . . . . . . . . . . . . . *19*
*The Universal Debugging Interface (UDI)* *20*
*Ldb* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *22*
*Pi and Blit* . . . . . . . . . . . . . . . . . . . . . . . . . . . *23*
*Other Efforts—the Testbed and LPdbx* . . . . . . . . . . . . . . *24*
*My Place* *25*
*Summary* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . *26*

**3. Building Blocks to an Architecture** . . . . . . . . . . . . . . **29**
*General Issues of Protocol Design* . . . . . . . . . . . . . . . . *30*
*Preliminary Definitions* *31*
*Client-Server Topology* . . . . . . . . . . . . . . . . . . . . . . *33*
*Client-Server Design Constraints* . . . . . . . . . . . . . . . . *33*
*The Demands of Developing Applications* *36*
*Monitor Resources* . . . . . . . . . . . . . . . . . . . . . . . . *37*
*Relationship between User Application and Monitor* *38*
*Dynamic Configuration* *39*

*Information Requirements of Dynamic Configuration* 40
*Namespace Architecture* . . . 41
*Monitor vis-a-vis Namespace* . . . 42
*Handler Call Interface* 43
*Namespace Modelling* 43
*Resources as Objects* 44
*Namespace Operations* 46
*Downloading Handlers* 48
*Debugging Semantics* 48
*Data Representation* . . . 49
*Debugging at the Event Level* 50
*Break Events* . . . 51
*Activity Relative Break Events* 51
*Summary* . . . 53

**4. Namespace Model . . . 55**
*What is a Namespace?* 55
*Handler Arguments* . . . 56
*Handler Execution* 56
*Handlers* . . . 57
*Environment* . . . 57
*The Default Environment* 58
*Namespace Model Closure* 59
*Handlers—In Practice* . . . 59
*Default Environment Handler Specifications* 60
*Finding Names* . . . 61
*Adding Names* . . . 62
*Removing Names* . . . 62
*Adding Handlers* . . . 63
*Removing Handler* 64
*Replacing Handlers* . . . 64
*Dynamic Modification—In Practice* 65
*Limitations of Handlers* . . . 65
*The Nature of Workspace* . . . 66
*Implementation Considerations* . . . 67
*Summary* . . . 69

**5. Specific Namespaces . . . 71**
*Root Namespace* . . . 72
*Device Namespace* 72
*Adding Hardware* . . . 76
*Typical Device Work-Handlers* 77
*Target Namespace* 78
*Entities* . . . 81
*Controlling the Program Under Test* 84
*User Services* . . . 86
*Breakpoints* . . . 87
*Break Event Modelling* . . . 89
*When Break Events Occur* . . . 94
*Setting and Clearing Break Events* 97
*Break Message Analysis* 99

*The Good and the Not So Good* . . . . . 102
**6. Driving the Model** . . . . . **105**
*Co-routines and Continuations* . . . . . 105
*The Co-routine Stack* . . . . . 108
*Storage Management* . . . . . 110
*Co-routine Structure* . . . . . 111
*Monitor Co-routines and Global State* . . . . . 113
*Models for Monitor Events* . . . . . 115
*Breakpoint Events* . . . . . 116
*Low-Level Model Types* . . . . . 117
*Low-Level Operations* . . . . . 118
*Interrupt Side-Effects* . . . . . 122
*The Global Message Queue* . . . . . 124
*Messages Received from the Outside World* . . . . . 126
*External Breakpoint Messages* . . . . . 126
*User Data Messages* . . . . . 126
*Namespace Engine Messages* . . . . . 127
*Dynamic Configuration of Hardware* . . . . . 127
*Summary* . . . . . 129
**7. Feeding the Model** . . . . . **131**
*Communications Namespace—Channel drivers* . . . . . 131
*External Namespace—The hole through the Wall* . . . . . 136
*Message Reception and Transmission* . . . . . 139
*The Switch Medium Protocol* . . . . . 140
*Summary* . . . . . 142
**8. Operation Decomposition** . . . . . **143**
*Target Independent Decomposition* . . . . . 143
*Notes on the Derivations Presented* . . . . . 144
*Operational Decomposition* . . . . . 144
*Referencing Namespace Workspace* . . . . . 145
*Namespace Model Derivations* . . . . . 146
*Special Env-Handlers* . . . . . 150
*The Break Handlers of Target Namespace* . . . . . 153
*Setting, Modifying and Clearing Breaks* . . . . . 155
*Controlling Program Execution* . . . . . 157
*Monitor Initialisation* . . . . . 160
*Summary* . . . . . 163
**9. Modelling Data Communication** . . . . . **165**
*Issues of Heterogeneity* . . . . . 165
*Communication Media Constraints* . . . . . 166
*Message Structure* . . . . . 167
*Transmission Packets* . . . . . 168
*Machine Independent Tag Values* . . . . . 172
*Concrete Message Format* . . . . . 175
*Message Construction and Extraction—Retrieval Functions* . . . . . 177
*Message Retrievals* . . . . . 177
*Argument Type Conversions* . . . . . 180
*Message Concretion* . . . . . 181
*Errors Concretions* . . . . . 182
*Response Concretions* . . . . . 184
*Summary* . . . . . 185
**10. Conclusions** . . . . . **187**
*Contributions* . . . . . 187
*Limitations* . . . . . 189
*Future Work* . . . . . 189
*Summary* . . . . . 191

Annexes


**A. Namespace Model Definition** . . . . . . . . **193**
*CD-ROM Contents* . . . . . . . . *193*
*The IFAD VDM-SL Toolbox* . . . . . . . . *194*
*A Description of Namespace Model Core* . . . . . . . . *194*

**B. Namespace Services** . . . . . . . . **195**
*A Description of Contents* . . . . . . . . *195*

**C. Model Drivers** . . . . . . . . **197**
*A Description of Contents* . . . . . . . . *197*

**D. Operational Decomposition of Core Handlers** . . . . . . . . **200**
*A Description of Contents* . . . . . . . . *200*

**E. External Namespace** . . . . . . . . **202**
*A Description of Contents* . . . . . . . . *202*

**F. Modelling Data Communications** . . . . . . . . **203**
*BNF Message Grammar* . . . . . . . . *203*
*Transmission Packets* . . . . . . . . *204*
*Machine Independent Tag Values* . . . . . . . . *205*
*Model Handler Argument Concretions* . . . . . . . . *206*
*Concrete Message Format* . . . . . . . . *208*
*Auxiliary Functions to Map Concretions to Model Values* . . . . . . . . *208*
*Message Retrievals* . . . . . . . . *209*
*Auxiliary Functions to Map Model Values to Concretions* . . . . . . . . *219*
*Message Concretion* . . . . . . . . *220*

**G. Environmental Requirements** . . . . . . . . **224**
*The Debugging Environment* . . . . . . . . *224*
*Communication Requirements* . . . . . . . . *224*
*Program Under Test Requirements* . . . . . . . . *225*
*Debug Agent Requirements* . . . . . . . . *225*

**H. Debugging Methodologies** . . . . . . . . **227**
*Debugging Techniques* . . . . . . . . *229*
*The Limitations of these Techniques* . . . . . . . . *230*

**I. The Rules of Operational Decomposition** . . . . . . . . **232**
*The Rules of Decomposition* . . . . . . . . *232*

**J. Glossary** . . . . . . . . **234**
*Technical Definitions* . . . . . . . . *234*

**K. References and Bibliography** . . . . . . . . **237**
*References* . . . . . . . . *237*
*Bibliography* . . . . . . . . *239*

# Chapter 1

# Introduction

Computing is a rapidly changing area of technology. Computing and communication technologies that were revolutionary twenty, ten or even as little as five years years ago are now regularly applied in end-user applications.

There was a time, twenty-five to thirty years ago, when inter-computer communication was slow, primitive, expensive and proprietary in implementation. Computers tended to be isolated from other systems; ideas such as data interchange were difficult to implement because of the closed nature of each system. However, over time and with modern advances in communications hardware, prices started falling, performance increased and moves toward standardisation of both hardware and software took place. As communications hardware became less expensive and more capable, networks started to proliferate and resource sharing using networked systems became a cost effective means of avoiding the extra costs of duplicating hardware that was otherwise not needed.

The integration of communications and computing technology is still only in its infancy. As the price-performance ratio of microprocessors increases and the same advances occur with communications technology it is reasonable to expect more applications pushing the paradigm, "the network is the computer" in fields that were previously unheard of.

### Origins of this Thesis

The background to the models and protocols presented in this thesis lie in project work done by the author during the final year of a first degree in Computer Science at the University of Waikato. The project involved communication between a host program and a target monitor. This work concerned the re-design and re-implementation of a software development tool, *Zmon*, that was used to load, execute and debug programs on target hardware. The host systems were Unix-based workstations which allowed the user to edit, compile and link user programs that could then be downloaded via a serial line to the target hardware. Communication between the Unix-based host program and target-based monitor allowed control of program to be tested. On each target machine software burnt into ROM provided monitor/debugger functionality allowing the user to:—

a. Examine and alter target resources such as memory and registers.

b. Download executable code from the host to the target via the serial line and run this downloaded code on the target.

c. Set breakpoints and single step through the program at a machine code level.

On the host machine a companion program, *odt*, handled all input and output to and from the target CPU so that messages would appear on the user's host terminal; keyboard input from the host terminal's keyboard would be directed to the target for processing. Downloading executable code was done by *odt* using the Motorola S-Record transfer protocol. *Odt* was the host interface to the *Zmon* target program.

*Zmon* has been quite successful. The software is relatively small in terms of code size (less than 40K of ROM). The task of locating, analysing and correcting erroneous behaviour (the task of debugging) of programs running on the target has also been made much easier for the programmer.

The *Zmon* project represents an in-house solution to a tool required for software development. During the design and implementation phase I became interested in more general solutions to the problems of debugging target systems. These problems can be summarised as:—

a. The abstraction of *essential* debugger functionality irrespective of the type of hardware being debugged.

b. Handling situations where a system's configuration may change during execution.

c. Providing mechanisms to debug distributed concurrent systems whose hardware requirements may vary widely as dictated by the requirements of the application.

I became convinced, through my experience with *Zmon* and discussions with my project supervisor, that *Zmon* and *odt* were insufficient to address the above needs. A completely new architecture would be needed to achieve these more extensive goals.

The undergraduate project work covered familiar ground to most people who develop software for such systems. Monitors and loaders are commonplace development tools. In the area of distributed systems, however, debugging tools which can handle the intricacies of concurrent processes executing on distributed machines, particularly in an embedded environment, are relatively scarce. These tools are often highly specialised for a particular hardware configuration.

## Distributed and Embedded Computing

Computer networks have been used to provide a means of sharing and pooling hardware resources. In the simplest case a distributed system might only allow for file sharing (through a file server) or remote printing (through some print server). Each service requires software that allows requests to be made by the requesting machine so that the remote entity can satisfy the request in some implementation dependent manner. The ability to share resources permits greater utilisation of machines which can offload tasks from busier general purpose machines. These distributed machines are specially dedicated (and optimised) to perform that task.

Alternatively, a service which a machine may be incapable of performing because that service is not physically available on that machine can be performed by sharing a remote resource. This concept of the remote service invocation is a critical component of a *distributed system.*

Distributed systems go beyond the capabilities of networked systems. In this sense CPU time is just as much a resource as is a printer or a disk. For example, a typical distributed system might need the immense computing resources of a supercomputer for the purposes of simulating a complex physical system. When it comes to visualising that simulation, however, the task's output is redirected to a front-end machine such as a graphics workstation for final display using a windowing system.

Distributed systems allow more than just the sharing of resources. A well designed distributed system can also offer increased performance through:—

a. better utilisation of resources better suited to particular tasks.

b. a simple means of scaling system performance.

c. a means of tolerating faults based around a concept of "No single point of failure".

As the price-performance ratio of computing has improved, the inherent flexibility of the microprocessor has become useful outside traditional areas of application. Many modern motor vehicles, for example, have computer-monitored engine management systems. Yet another example of the application of microprocessors to the field of process control is growth in the availability of "smart appliances". Such systems are known as *embedded systems.*

The concept of an embedded system need not only apply to a single isolated "unit". Telephone switching networks and banking systems are also examples of embedded systems. Such a system relies on communication between computer systems over some distance. Thus *distributed embedded systems* can also exist.

## The Task of Debugging

Programming is a non-trivial task. No honest computer programmer is ever likely to claim to be able to produce any non-trivial program which executes according to specified requirements first time.

How many software engineering products fail while in use because of bugs? How many civil engineering products fail because of faults? Why are software projects unable to be engineered with the same success as civil engineering projects such as the construction of dams and bridges?

Civil works differ from software engineering projects in one very important aspect—the ability to test. By the time a civil engineering project is actually completed, extensive modelling in the form of, for example, wind-tunnel testing or seismic testing has been applied to the prototype model and validated *before* the project has been completed. Any weaknesses in design prior to construction can be re-engineered until required tolerances are successfully attained. Engineering has the domain of the physical world as its guide, a background of theory and physical laws that place definite constraints on the construction of the project. Component testing at each stage of the design and the construction process, ensures all tolerances are met and minima exceeded. In many cases, reuse of commonly available parts and materials known to meet the required standards can be relied upon. A civil engineer has a background of knowledge and experience that suggests that provided environmental tolerances are met, the project will meet operational requirements.

Civil engineering works have to be well constructed. Human lives depend on bridges that do not self-destruct in a light breeze and dams that do not burst in times of severe flood. Likewise, many embedded systems also have a *safety-critical* component. The timely operation of a cooling valve controlled by a computer control system that monitors a nuclear power plant could avoid a critical melt-down of the nuclear reactor; thus potentially endangering human lives.

It is suggested that the following question needs to be answered—do the same testing, prototyping and modelling facilities exist in a software engineer's environment? Should they be required?

Compared to the civil engineering paradigm, when does the software engineer carry out quality assurance (QA)? Even though some prototyping and modelling tools exist, in most cases the testing is only done *after* implementation. The lack of reuse of common (tested) components also means that much re-invention of the wheel has to be done and this has an effect on re-introduction of errors. Only relatively recently (in the last five years) have model testing and building tools become available.

Software complexity arises due to the multitude of components that a software system is composed of—compared to, say, a bridge. The "interfaces" to attach parts of a bridge to one another are simple and regular. Software module interfaces are far more complex and less regular than objects such as girders. Testing and verifying software modules in isolation to high levels of reliability

is difficult and time consuming. This is especially the case where non-determinism is involved. Using individual modules in combination with each other in an integrated system makes testing even more difficult. Software construction is at least an order of magnitude complex than civil engineering projects because of modelling and the module testing problems.

While the area of software testing and debugging has remained much of an informalised topic (Myers [Mye79]), tools for debugging programs have steadily improved in usability. Debugging tools are far more advanced now than comparable tools two or three decades ago. Once programmers had to debug with little more than memory image dumps, stack traces and disassemblies of executable images. Today, debugging environments allow for GUI-based, source-level tracing of software being debugged.

### The Debugging Environment

The debugging environment can be split into two separate components—a component that allows a user to *interact* with debugger capabilities and a component that allows the debugger to perform the actual task of debugging. The debugger can be considered an application much like any other. The Arch-Slinky diagram presented below [Gram96] provides a model that relates tasks performed by an application to the user interface that allows a user to enact that task.

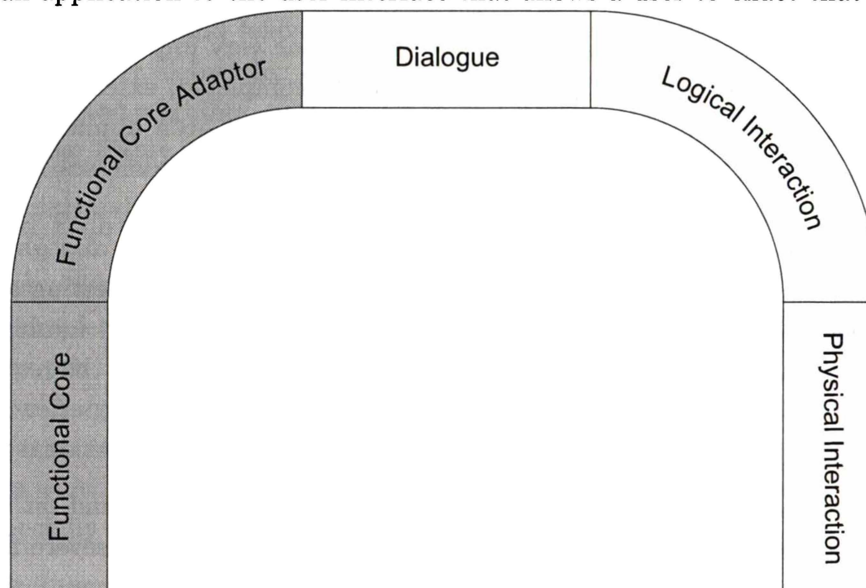



Fig. 1.1 *The Arch-Slinky Diagram—illustrating the components of a debugging system*

With the advent of distributed and concurrent computing systems, the nature of debugging has changed. Previously, the software being debugged and the debugger itself existed on the same machine. While debugging a program on one machine is never trivial, debugging in a distributed environment where programs may execute on any computing node within a network is inevitably more difficult because parts of a program may be executing concurrently on many processors. How does debugging of such systems take place? Where does the debugger lie in such a system?

### Debugging Instrumentation—Tools of the Trade

Consider a process control system and the steps required to examine its progress and alter its behaviour. Studying process control flow may not be debugging as such, but it is monitoring, a fundamental component of the debugging process.

Consider a chemical processing plant, a monitoring system will take inputs from a variety of sensors situated in key areas of the plant. These sensors allow the reading of various kinds of

information needed for monitoring and control of the process. The plant also contains actuators that take control information and modify process behaviour. For example, a given actuator might raise or lower the temperature of a heating element or it may be a valve that regulates the flow of constituents. Central to process control is the human operator or possibly some automated system that can make decisions based on data returned by the sensors to then effect changes by controlling actuators.

There are two critical components in this control system:—

a. The interface that allows the control information to be obtained.

b. The actors that actually control the plant process.

The importance here is that the essential control services in any process control system *are* these interfaces.

A distributed debugging system should be treated the same way as a process control system. It is the "core" debugger services that are important. This thesis is about modelling the target process interfaces to a debugger.

## Thesis Domain

The human user of the debug utility inevitably requires an interface to the running processes in terms of tools which are going to be useful to a programmer. As in the Arch-Slinky model, this thesis concerns itself with the *functional core*, *functional core adaptor* and its interface to the user dialogue. This dialogue component shown in Figure 1.1 is referred to in this thesis as the *debug agent.*

## A Model and a Protocol

This thesis describes the design of *protocols* and *server architecture* for debugging distributed programs in embedded systems. This concentrates on protocol and server design—the *instruments* through which debugging is made possible.

The debug server and interface protocol are central to the implementation of *any* debugging system. Without the debugger server and protocol, the debug utility, no matter how ingenious, is useless. Concentrating on both the protocol and server design, the major emphasis of this thesis therefore becomes that of the design of the *instrumentation* of a system for debugging. What is the minimum working set of instruments required to debug? How can the instrumentation minimise its influence on the program under test? How can this influence be formally defined?

To model a solution to any problem, it is desirable to use a notation which describes the solution unambiguously. This is particularly important when considering an abstract hardware-independent model such as that presented here. What is needed in order to clarify thinking is a formalism, in this case a formal modelling language. Of the formalisms currently available, *VDM-SL (Vienna Definition Method Specification Language)* was chosen for the following reasons:—

a. It has a rigorous ISO standardised mathematical definition.

b. While being a very well defined notation, VDM-SL is similar in outward appearance and meaning to "traditional" programming languages such as C++, Ada and Modula-2. This means that there is no major "paradigm shift" for many software engineers in order to use it.

c. It was also chosen because reference materials readily existed on campus and that some faculty members had suggested its use.

VDM-SL is capable of modelling more than just structure and implicit operation behaviour. Modelling using the implicit subset of VDM-SL is the same as making assertions on the input state of arguments and the output state of operations and functions. Implicit modelling specifies *what* should happen without making commitments as to *how* it should happen. VDM-SL is also *multi-spectrum*, it can also be used to derive an *explicit* specification suitable for translation to some target programming language via the processes of *data reification* and *operational decomposition*. The IFAD VDM Toolbox software [Lar98][IFAD99] has been used to syntax and type check the models derived in this thesis. The toolbox is compliant with the ISO VDM-SL standard in terms of syntax but there are some semantic variations between both the standard and the toolbox that are described in [IFAD99].

For the sake of brevity, the majority of VDM-SL model definitions are presented in the Annexes with only major definitions being given in the main body of the thesis.

### Summary

This chapter has introduced the original inspirations for the work along with an introduction to the problems of distributed systems and the non-trivial task of debugging a program.

Chapter 2 examines the methodologies and techniques used by users to debug programs. These are considered further in the context of distributed systems debugging with an examination of the problems posed by such systems. The chapter concludes by contrasting several currently available solutions with the requirements of an "ideal" distributed debugging tool.

Chapter 3 considers design goals and constraints of the debug protocol to be defined in later chapters. An introduction to the proposed model and protocol is given.

Chapter 4 formally describes the namespace model that forms the core of the solution proposed by this research. This chapter develops the namespace model and defines the minimum working set of operations on the model required for closure.

Chapter 5 defines namespace environments that form the debugging services component of the monitor server. In particular, the device namespace and target program namespace services are discussed in detail.

Chapter 6 defines the functional core adaptor component of the monitor server that is required to support the namespace model. The support environment describes co-routines used to pass messages between both the debug agent and monitor server but also between the co-routines and the attached namespace model. The "low-level" functionality, defining the storage model, message queues and continuations required for the support of co-routines is also defined.

Chapter 7 is devoted to describing the communications interface between monitor server and debug agent. The chapter describes both the driver interface and the dialogue interface used to establish connections between monitor server and debug agent.

Chapter 8 builds upon the specifications in Chapters 4 through 7 and presents operational decompositions. The issues of data reification are left to the potential implementer of the specification.

Chapter 9 considers the issues of message concretion to and retrieval from a communications medium in this instance modelled around an ISO-8801 Ethernet frame. This chapter presents functions to convert to and from abstract data forms as one might do when messages are sent by or received from some network interface.

Chapter 10 analyses the contributions of the research and scope for future work with reference to the Arch-Slinky diagram presented in this chapter.

# Chapter 2

# Looking for an Interface

"Furious activity is no substitute for understanding."
*H. H. Williams*
*Oakland, California* [Ben88]

### What is Debugging?

*Debugging* is the process of locating, analysing and correcting faults in a program. *Faults* in this case may be considered to be the same as errors, accidental conditions that cause a program to fail to perform to its specification. The term *bug* is often used colloquially to describe the occurrence of such a fault or error.

Debugging *is* a difficult task. There has been little study (Araki [Ara91] and Cheung [Che90]) into methodologies of the process associated with debugging programs. Debugging is often just considered to be just another stage of the development cycle, much like the specification, documentation, design and implementation stages of a software project – albeit less rigorous and formalised when compared to other stages of the development cycle. Unfortunately, however, debugging and testing of software takes up a considerable amount of time within software development cycle—time that often translates to delays in software releases.

### The Problems of Debugging

It has been the experience of the author in a software development environment that testing and debugging efforts benefit most from a software system whose requirements are well defined and whose implemented components are well specified. The efforts of testing and debugging are closely linked—*testing* is the process of determining whether a component behaves according to specification. Having established that a fault exists the task of debugging must take place. If a software system has not had its requirements explicitly elicited and its behaviour well specified the term correctness ceases to have meaning. Debugging such a system then becomes a matter of perception of what *might* be wrong rather than what is *known* to be erroneous behaviour.

It must be conceded however, that even in the most perfect of worlds where both system requirements and specifications are well founded, faults will, at least initially, still exist. The testing and subsequent debugging of such systems will be easier though. Well specified systems are a step toward systems that are easier to debug. While important in the context of the software development life cycle, the specifics of requirements and system specification are not considered any further in this thesis.

Once the testing process has established the presence of faults, the process of determining the cause of the erroneous behaviour and correction of the error takes place. Bugs are a manifestation of various symptoms—a sign that behaviour is not as expected. These symptoms may give a hint as to what the disease might be. The fault correction phase is then about "curing" the disease.

Finding bugs is best accomplished with knowledge of what *should* happen. This knowledge should only be gained by analysis of requirements and detailed specification of system behaviour.

Why is debugging difficult? Myers [Mye79] and Cheung [Che90] identify reasons why debugging is potentially the most difficult component of the software life-cycle:—

(1) For those programmers who take considerable pride in the code they produced it is psychologically difficult for that programmer to accept that the code they have written may have faults.

(2) The activity of debugging is mentally taxing. This is usually compounded by work induced pressures to fix the fault within some given time constraint or deadline. Often, frantic, intense debugging effort does not guarantee that the bug will even be found, let alone corrected.

(3) Given the often large state space of software systems, faults can be extremely difficult to localise. The bug could be literally anywhere in the programmer's code. With the advent of structured programming practice and modular programming languages this is less of a problem but faults still occur. Contrast this fault finding scenario with that of a mechanical system such as a car, a stove or a hot water kettle. Which component do we blame when a car stalls on a cold day? Of the two components, the engine is more likely to take the blame not the car radio.

(4) A programmer often gets locked into a view of what is believed to be the suspected problem even though there may exist many other reasonable options. From a psychological standpoint, any doubts the implementer has during development, no matter how unfounded often manifest themselves as being "suspect" when in fact they are not.

Debugging can be divided into two phases, fault localisation and fault correction:—

a. Fault localisation may involve the postulating of hypotheses and their subsequent proof of truth or falsehood. This may often involve the use of debugging tools.

b. Fault correction involves the steps taken to remove erroneous behaviour from program code. It is typically an offline operation requiring a widely varying amount of time and effort.

The term *hypothesis/hypotheses* in this case implies theories about characteristics of the program code that may cause erroneous behaviour. These hypotheses are based upon qualities of the program code such as application domain, formal specification and program dynamics as well as theories on the classification of errors, their cause and possible modifications to correct the errors [Ara91]. Refer to Annex H for a description of debug methodologies as described by Araki [Ara91] and Myers [Mye79].

## Debugging Tools

When present and in use, debugging tools represent the "tangible" component of debugging. Debugging tool usage is generally considered to be part of dynamic analysis. Therefore, part of the job of a good debugging tool is to assist in the process of hypothesis verification.

Debugging tools that are provided as part of a development environment for host based programs could be viewed as extensions of the underlying operating system to the subject program with the debugging tool(s) acting as the "broker" between them. The debugger is a special kind of operating system program—it allows the programmer the capability to *control* the execution of the subject program being debugged. The debugger must handle all situations that arise, such exceptional circumstances as trace traps, breakpoint events and normal/abnormal termination. When these situations do occur then control must be returned to the debugger instead of the operating system. For example, under the Unix operating system the debug utility runs as a parent process with the

debugging subject (the program-under-test) running as a child process—the debug utility parent has special privileges granted to it that allow the human user to inspect the child's process space and control its execution.

Debugging tools exist on numerous computing systems in a variety of forms that may be simple or complex. Tools on Unix workstations such as *sdb*, *gdb* and *dbxtool* provide powerful debugging environments when debugging programs developed for Unix hosts. These tools typically allow debugging to take place at a relatively high level of abstraction, in this case at the source level, from a statement by statement view. Compile time debugging information is used by the debugger to provide symbolic access to scalar and structured types as well as the source code on a line by line basis.

According to McDowell [McD89], a good debugging tool for a single processor machine should provide tools to:—

a. Read and write to memory.

b. Set and trap breakpoints.

c. Trap program exceptions.

d. Provide trace and single step ability.

e. Trap memory accesses.

Debugging tools of even a rudimentary form did not exist until the late 1960s. The primary reason for this was that CPU hardware did not have the capabilities to support such tools. For a simple debugging tool to exist, hardware support from the CPU has to provide a software trap/interrupt/exception mechanism so that control can be returned to the debugger when a program fails or when breakpoint events or single step interrupts occur. Interactive debugging tools have only really existed since the early 1970s. Prior to this, the primary means of debugging was the use of debugging libraries which were directly linked into executable code to provide information on the executing program. These debugging libraries sent information to a terminal or line printer in what amounted to an execution trace. The problem with this approach was that unnecessary modification of the source code was done. Debugging libraries had to be compiled as part of the generation of an executable image. While this could work well, there were times when the debugging library code could actually mask out the bug.

In a similar way, debugging tools that exist on development hardware systems such as monitor programs *are* the operating system of the target hardware—they are bootstraps to the target hardware representing a minimal level of functionality to allow development work to proceed. The example of *Zmon/odt* referred to in Chapter 1 represents a member of this category. The monitor/debugger programs that exist in such configurations normally provide primitive operating system facilities such as loading (or downloading) programs and simple terminal input-output for interaction.

## Debugging Programs on Single Processor Machines

Debugging programs on single processor architectures represents an area to which most of the programming community has been exposed at one time or another. Debugging tools for various operating systems such as Windows NT, MS-DOS, Unix, VMS, etc are commonplace and often assist in the process of locating and correcting faults. Generally the debugging phase of a software project is a combination of the tools, the techniques and methodologies that have been mentioned.

Most of the time, albeit with considerable effort, the techniques work and the code gets debugged successfully.

However, the subject of this thesis is not about debugging uniprocessor machines. This thesis is about models that provide a mechanism for debugging distributed programs on embedded systems. These distributed machines may vary widely in architecture and performance. In refining the research topic it must be acknowledged that technologies for debugging uniprocessor machines are still not yet perfect and need to be further investigated.

### Traditional Uniprocessor Debugging Architecture

What is the difference between distributed and non-distributed debuggers? The answer lies in where the debugging takes place. Traditional debugging architectures view the debugger and debugging subject as processes that are executing in the same *physical address space.* Depending on the implementation, the debug utility and program-under-test may have different *logical* address spaces but they always share the same physical address space. Debugging is somewhat less difficult from a conceptual point of view when the debug utility and program-under-test share the same address space due to state being readily accessible. Such debugging tools are far easier to implement because such programs tend to have a more simplified view of the environment they debug—they assume only that hardware and only that configuration. This static view of the environment is unacceptable if a distributed debugger is to be implemented because environments and configuration vary considerably between processor nodes.

Traditional debugging architectures have a "traditional" view of program behaviour. Communication between processes is allowed, typically by means of shared memory or named pipes.

When writing programs for parallel and distributed systems the notions of common address space for communication no longer apply. For inter-process communication the idea of shared memory still operates but for intra-processor communication there is no shared memory medium to communicate—just some pieces of wire.

### Debugging Parallel and Distributed Programs

"The first step in fixing a broken program is getting it to fail repeatably."
*Tom Duff*
*Bell Labs* [Ben88]

Parallel programs often exhibit non-deterministic behaviour, getting repeated failure that can be used to analyse and correct faults can be extremely difficult. Cheung [Che90] and Joyce [Joy87] cite the following reasons why parallel and distributed debugging is so difficult:—

(1) Maintaining precise global state is impossible. Global state is difficult to collect due to different latencies in various types of communications devices and the varying communication rates of information acquisition due to system loading. Total ordering of such events is impossible because there is *no* global clock. A partial ordering of events using a "happened-before" relationship to order messages (sending a message temporally occurs before its reception by another process) is generally the best that can be done.

(2) The state space for parallel and distributed systems is large. Interactions between processes can make debugging difficult because of the number of events that can be generated and the notion of ordering that is required to make their interactions meaningful.

(3) Interaction between multiple asynchronous processes is complex. The faults that often occur in such systems are sporadic in nature. These faults may be due to race conditions or

synchronisation problems. Relative timing between processes, especially in a distributed system, can also cause subtle faults to surface.

(4) The communication medium is non-deterministic. As explained above, this limitation makes timely state information difficult to acquire. Error latency, or the time period between the occurrence of an error and its discovery, is a consequence of this problem. Often the originating error gets masked by consequentially propagated errors.

(5) The *Probe Effect* is a product of Heisenberg's Uncertainty Principle applied to computing. The activity of attempting to gain information to isolate an error in a distributed system by the use of probes such as breakpoints and traces can modify the original behaviour that the "probes" were there to observe. The inserted probes may mask the occurrence of the fault or cause new faults to occur. The faulty behaviour may be caused by subtle timing problems that are removed (or generated) when intrusion by the debugger takes place.

## Distributed Debugging Architecture

In a traditional single processor machine, debugging is relatively straight forward. The subject program and the debugger exist on the same machine. All resources needed by the debugger to enquire about the state of the subject program are available. The debugging process could be described as being tightly coupled.

Distributed systems exhibit decentralised control and loose coupling. Debugging such systems however, calls for a centralised master-slave approach. Given that a distributed system consists of interacting processes that execute on different machines, there is a need to snapshot that interaction. Without having a centralised agent to use this information, debugging would be impossible. Debugging a distributed system needs a global perspective of software behaviour, the local view from each processor is not sufficient. Distributed systems are about "programming in the large"—the distributed debugging architecture is an extension of the traditional debugging architecture where not only the local state of a program is required, but also the relationships and interactions of the system as a whole are important. Debugging distributed systems needs a centralised agent to debug a set of distributed client processes that are executing on many machines. A centralised debug agent is beneficial in this case—it provides a frame of reference.

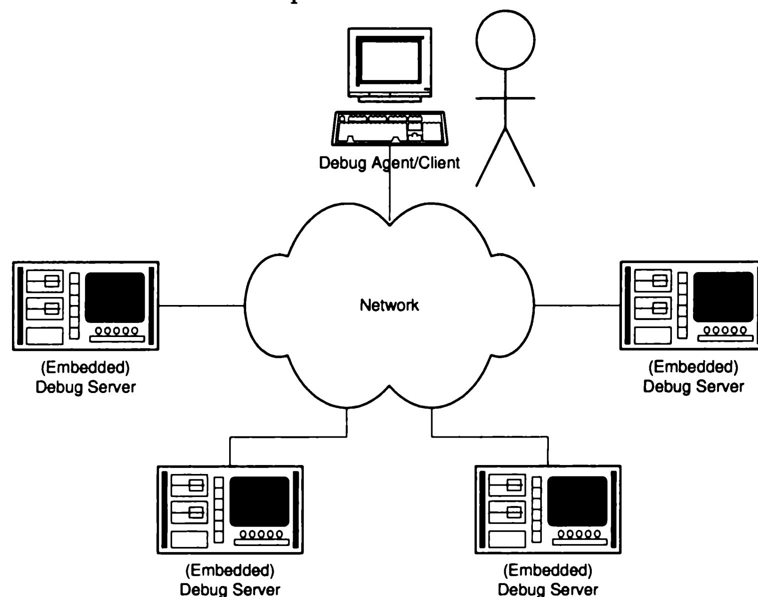



Fig. 2.1 *Distributed debugging architecture*

The debugging architecture implied by this discussion is closely related to that of the client-server model of computing. Each target machine in the distributed environment is a server which accepts and acts upon requests and enquiries made to it by a debug agent acting centrally to the debugging process. The notion of having a centralised agent is entirely arbitrary, there is no physical meaning that should be attached to it. Any machine in the distributed environment can run the debug agent, it does not matter where it is, it is just sufficient that it exist to provide a reference frame.

### Embedded Systems and Debugging

Embedded systems are becoming the largest consumer of microprocessors today. A growing number of microprocessors are being sold to the embedded systems market for the purposes of process control in a wide number of application fields. The variety of applications using microprocessors is growing due to the ready availability of cheap processing power provided and the decreasing cost of support hardware such as memory and signal processors. Some examples include: anti-lock brakes in cars, subject range determination in auto-focus camera equipment, missile guidance systems and many industrial processes such as monitoring systems in nuclear power plants. Kündig [Kün86] defines an embedded system as one that has the following characteristics:—

a. An embedded system is an electronic system embedded within a given plant or external process. The external process may involve humans performing supervisory or parameter setting tasks. Usually, the external process is an on-going activity which does not allow restoration of previous state prior to failure. Furthermore, embedded systems exhibit the property of *real-time* response—the correctness of the system is based not only on the action, but also whether that action occurred within a prescribed period of time.

b. The external process and embedded system are coupled through suitable sensors (such as a sound or temperature transducer) and actuators (such as a stepper motor). There may be real-time requirements that specify timing relationships between certain input and output signals.

c. Sensors and actors, as well as the processing systems themselves may be distributed over a wide geographical area. The processing systems may be tightly or loosely coupled.

d. Reliability is a strict requirement of such systems. Often, embedded systems are known as *mission-critical* systems. The failure of such systems could potentially result in a loss of life. Such a system may use redundancy in order to fulfil reliability requirements. The construction of the system may be made more robust because it will be subject to environmental conditions such as extremes of humidity and temperature. Size, weight and power consumption requirements may also exist.

e. Processing elements may not be homogeneous. Some processors may be highly specialised for a particular task such as signal, numeric or communications processing. The processing system can be optimised to the task required of it and no more. Extremely high processing power or specialised computer architectures may be required for applications such as computer vision or speech recognition.

f. Depending on the application, an embedded system may exhibit highly modular design of both hardware and software. *Reconfiguration and expansion of a system could take place at run-time.* Both hardware and software could be added or removed while the embedded application executes.

All systems have performance requirements—in this manner, all systems can be viewed as being embedded. For example, electronic fuel injection, have certain timing constraints that are precisely

controlled by the embedded systems hardware and software. Debugging is, however, in its most typical form, an invasive activity. When timing of a critical nature is involved however, intrusion by the debugger is unwanted. Often a debugger is more of a hindrance than a help in such cases since an intrusion by the debugging software can "mask out" critical timing problems.

### Defining a Common Interface

All past efforts at producing viable debugging tools for distributed development have always had a common noble goal—the ability to easily analyse distributed system behaviour and to allow the easier comprehension of complexity inherent in such software systems. Such goals can be achieved in a number of ways as far as the programmer is concerned. Debugging tools exist at a number of levels of abstraction. The levels are not discrete but form a continuous line. For example, some of the debugging tools described here use dedicated hardware to capture significant sequences of CPU instructions or bus signal traces as defined by the human user. At the other end of the scale, some debugging tools are based upon high level language source which is compiler supported or some object level of interaction between other objects requiring specialised tools to determine object creation, object destruction and messaging (IPC) relationships between objects.

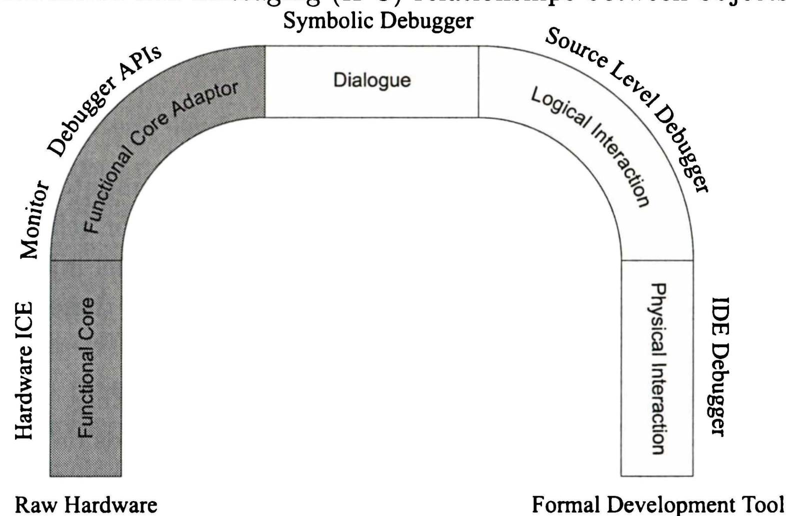



Fig. 2.2 *Debugging interface levels*

To imply that a particular level of abstraction for debugging is better than another is incorrect. What is correct to say is that the type and context of the problem will have some bearing on what tools might be used to localise the cause of the problem.

### A Taxonomy for Classifying Embedded Distributed Debuggers

Applications that are designed for distributed embedded environments exhibit particular requirements as described earlier in this chapter (Kündig [Kün86]). Because the debugging tool (the monitor) *is* an embedded application it seems reasonable that *debugging tools* for the development of such applications should exhibit similar characteristics:—

(1) Distributable—An instance of the debug server (the monitor) must exist in a loosely or tightly coupled environment on each of the target processes over which the program-under-test is distributed. This is a fundamental requirement!

(2) Reliability—It is assumed that the debug server (the monitor) in its distributed form *is* reliable. Once again this is a fundamental requirement. A monitor without inherent reliability and robustness is useless.

(3) Minimal Interference—The cost of inserting probes into the application must be minimised to lessen the likelihood of the probe effect.

(4) Heterogeneous—It should not matter whether the distributed application is developed for a heterogeneous processor environment. Debugging should be possible in mixed processor environments.

(5) Configurable—Both embedded hardware and software evolve over time. The debugger (monitor) should be able to evolve over time so that the capability to debug is not compromised.

Properties (1) and (2) are fundamental design properties and will be assumed for the remainder of this and subsequent chapters. What remains from this list ((3)–(5)) is a strict criteria for the evaluation of distributed debuggers. However, one further attribute must be considered; the capability to debug embedded applications—there is a need for the ability to debug embedded applications from the standpoint of a raw target platform without operating system support.

Classification of distributed debuggers for embedded systems is based on the following 4-tuple:—

(1) Embedded Applications Support

(2) Minimal Interference

(3) Heterogeneity

(4) Configurability

### Embedded Applications

Embedded applications form a large part of the dark matter of computer systems currently in existence. The performance and increased safety of many modern motor vehicles is due in part to the use of embedded systems for engine management duties such as fuel injection. The anti-skid brake is another example of embedded computing. Removal of either of these could be detrimental to the performance or safety of the motor vehicle.

Embedded applications often exist in the realm of *real-time* computing. Real-time computing refers to the concept where the correctness of a system not only depends on logical correctness of computation but also upon *when* the result is produced and subsequently acted upon (Stankovic [Sta88]).

The real-time definition has no absolute time scale that separates one application from being real-time from another that is not. There is often a misconception that real-time means fast processing, it does not (Stankovic [Sta88]). Many embedded applications actually execute on hardware that by today's standards is not particularly fast.

All applications then can be described as being embedded, but with differing timeliness requirements for responding to events. An operating system kernel executive is different to a windowing system which is in turn different to an interactive voice response (IVR) telephony application. The users of each of these applications might appreciate that response in each is as rapid as possible but the fact remains that timeliness requirements of each are an order of magnitude apart with the OS kernel requiring the hardest deadlines and the IVR application having the longest deadline in relative terms.

For applications executing under some stable user oriented operating system, the current debug environments are sufficient for the domain of application. Where applications need to be developed from scratch on bareboard development systems an operating system cannot be assumed. In these situations the monitor must act as a low level bootstrapping tool. Another facet of tools for debugging distributed embedded applications is the ability to bootstrap applications where no underlying operating system facilities are available. This includes process/thread management, memory management and any messaging capabilities.

### Minimising the Probe Effect

There are three approaches to the minimisation of the probe effect:—

(1) Ignore it—many non-real-time debugging systems make the tacit assumption that the debug server will not affect the program-under-test.

(2) Minimise the Effect of the Debug Server—implement the debugging system in such a manner that the debugger's affect on the program-under-test is minimal.

(3) Use Logical Time—where the debugger halts the execution of a process that is subject to time-out, logical time can be used so that a time-out can only expire when the execution of the halted process is resumed.

Minimising the effect of the debugger can be achieved by essentially two means. The first is to use hardware to passively monitor the processor bus mimicking the target processor's bus and instruction cycles. At the other extreme, the debugger calls should be "as fast as possible" so that the time taken for a fault to manifest itself is longer than the maximum debugging call servicing period.

### Concurrency Support

Concurrency exists in two levels of a distributed system. There is the pseudo-concurrency between processes executing on a single processor node and there is the "true" concurrency between processes executing on different processors simultaneously. Add to this the concept of inter-process communication and the asynchronous nature of this activity makes debugging *very difficult* [Che90].

In a distributed system, coupling is loose—there is no notion of global time since each node in the system has its own time reference. Implementing a global clock to which all nodes can synchronise is a difficult problem, even more so when the system exists over a wide physical area [Che90]. Not necessarily all machines run at the same clock speed, causing rates of program progress to be vastly different on various machines in the network. Communication between nodes in the system is subject to the non-determinism of the network, arbitrary time delays can take place depending on network technology used.

A large component of the problem of debugging concurrent processes is the need to ensure correct event ordering and some means of controlling the rate of flow of events—slowing down or stopping the clock when convenient so that interactions between processes can be examined in detail.

Why is global state important in the debugging of distributed systems? The reason for the importance of global state is that it is a property that allows the measurement of progress (or the lack of it) and thus enables the detection of stable conditions such as termination or deadlock.

Capturing global state in a loosely coupled distributed system is difficult due to non-determinism of the communications medium and the lack of a global clock. Progress in such systems takes place at differing rates, communication between processes on different (or the same) processors form the points of synchronisation. A major problem caused by delays in communication is the fact that a total ordering of events in a distributed system (composed from all events that occur at particular nodes) is impossible. The best that can be achieved is a partial ordering of events. Obtaining consistent global state is complex. Global state consists of states of a process local to a processor node but also messages sent from one process to another process (possibly on another processor) that are in-transit. Algorithms used to determine consistent global state function by having processes record their own local state. Additional "control" messages to other processes to

record their local state with some acknowledge count mechanism to determine when a "round" of global state has been completely captured (all local states for each process/processor have been attained). There are a number of algorithms that have been developed for distributed systems, for example Chandy [Cha85] and Mattern [Mat93]. Related work on the theory of logical clocks is covered in Lamport [Lam78] and Raynal [Ray92].

Support for distributed process debugging requires that significant states or events are captured as they occur. This is generally accomplished by the use of breakpoints:—

a. State Breakpoints—the process(es) will be halted based upon whether a global or local assertion becomes true or a particular statement is executed.

b. Event Breakpoints—the process(es) will be halted based upon the detection of events. This may be based on a state based breakpoint or through other means such as hardware. Events can based upon the multiple set of events occurring, a specified sequence of events or the occurrence of a single event.

Breakpoints also may have an influence on the entire program-under-test when a breakpoint occurs. Either one process, a group of processes or all the processes that make up the application may be halted.

### Support for Heterogeneous Environments

The heterogeneous distributed systems domain is one area that has not been well addressed by previous debugging research. Past distributed debugger/monitor efforts have been limited to closely coupled homogeneous distributed systems or multiprocessor machines. With appropriate compiler/cross-compiler support on the host, why should there be any limit on the variety of different targets of which a distributed system could consist? It is asserted that there should be no restriction on the type of hardware that can be used to build a distributed system. It does not matter how the individual processor addresses or represents data in its memory or register file, the protocol should abstract this.

The problems of handling heterogeneous machines in a network are well known and a variety of solutions exist to cope with the problems of differences of byte ordering between systems. The issues of hardware design and software portability with respect to network data representation are discussed by Cohen [Coh81] and Kirrmann [Kir83]. Most debuggers currently in existence are written for particular hardware platforms. The distributed systems domain has problems with how to meaningfully exchange messages between processor nodes. For example, the TCP/IP protocols define all datatypes with the most significant byte at the lowest address, a so called *big-endian* byte ordering [Coh81]. The fundamental datatype for TCP/IP protocol suite is the *octet*, an eight bit quantity. Other machines have a native data representation has the least significant byte at the lowest address (*little-endian* byte ordering [Coh81]).

What happens when data exchange has to be considered—for example when remote procedure calls are made from one machine to another? Sun Microsystems defined a data representation protocol known as XDR (eXternal Data Representation) for use with their remote procedure call software. XDR defines various datatypes such as thirty-two bit integers, sixty-four bit floating point numbers and character strings. These datatypes are converted into a common agreed upon format that can be sent over the network and upon reception converted into a native format that preserves meaning on the remote machine. Other protocols that perform similar operations are Apollo's NDR (Network Data Representation), Xerox's Courier and ISO's OSI Level 7 data representation syntax ASN.1 (Abstract Syntax Notation 1). For meaningful debugging of heterogeneous distributed systems similar mechanisms are needed to handle the large number of different processor types.

Ultimately, support for heterogeneous systems is dependent on whether the debugger is event oriented or state oriented. Event level debuggers provide information at a higher level of abstraction and therefore do not need to consider the issues of data representation. Any debugger that is state based will need to consider such architectural decisions.

### Configurability

Distributed processing environments are dynamic. Nodes are added, hardware is added to individual nodes and newly developed software is loaded onto nodes for testing. This is in contrast to desktop operating systems where the overall physical environment is static.

A monitor for a node *must* be capable of handling changes in both hardware and software configuration. The monitor should have as much right to access newly added hardware resources as the embedded application. If the monitor is notified of such changes then ultimately debug client needs to notified of this change by the debug agent. As so called "hot-insertion" devices become more widely used, the ability to adapt to configuration changes will become more important.

The following sections describe the current state of the art in terms of tools based upon how the criteria above and how the work of this thesis is related to these efforts.

## Hardware Based Monitors and Debuggers

The work of Tsai [Tsa90a][Tsa90b] at the University of Chicago is a loosely coupled distributed system. It was primarily developed for monitoring but can also be used for the purposes of providing information useful for debugging.

Each node in the system is associated with an *Interface Module* connected to a *Development Module*. The interface module is a hardware based solution designed to mimic and record the target processor's actions when specified triggering conditions are satisfied. The purpose of the development module is to provide the user a means of initialising the interface module, to control the recording of execution histories (record the instruction stream) and to perform post-processing on recorded stream.

Triggers to initiate and shutdown recording of instruction traces are based upon the detection of specific signals occurring on the target processor's bus. Bus control signals can be used to trigger events that describe the change from user to kernel modes of operation. The data and address buses can be used to determine whether a particular instruction is being executed or that a write to a particular memory address has occurred.

Although not specified, it appears that each target node needing monitoring or debugging requires a separate interface module and development module. This is because both modules are coupled by control and memory buses to each other. The interface module also passively exists on the target processor's bus so that instruction traces can be recorded.

Tsai's monitoring system does not address the heterogeneous domain. It could be argued that because the monitor operates at the event level it does not have to consider the problems of data representation. In practical terms the interface module must be redesigned for each target processor. Different target hardware implies the redevelopment of both interface and development modules.

The development module allows the user to specify events of interest that might be of use in debugging a particular scenario. There is no other explicit configuration management mechanism.

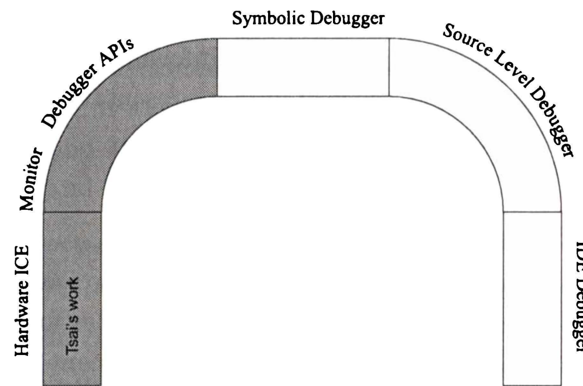



Fig. 2.3 *Tsai's monitoring system*

### The Test and Measurement Processor

The *Test and Measurement Processor (TMP)* research of Haban and Wybranietz [Hab90] is conceptually similar to the work of Tsai. Usage was also for similar reasons—monitoring and observing the performance of distributed applications while aiming to limit the cost of monitoring. By allowing behaviour of distributed applications to be visualised the front end monitoring tools have been used for analysing faulty behaviour and a debugging system has been also built on top of the monitoring tools provided by TMP.

The target architecture around which TMP was implemented is loosely coupled with each node containing its own processing hardware, local memory and local peripheral (including communication hardware). Each node has associated with it a TMP node responsible for the development node upon which the developed distributed application executes. The TMP node behaves in a similar manner to the monitoring system of Tsai, it is responsible for the monitoring and recording of activities occurring on the development node. Where TMP differs from Tsai's monitoring architecture is that TMP nodes can be connected together and can co-ordinate with other TMP nodes or a centralised monitoring station. While TMP nodes can be connected to the same network as the host, the absolute minimum interference can be guaranteed by having the TMP nodes connected in a private network completely separate from the host network. A TMP node is capable of local processing (including display) of captured data and events from the development node but it is more likely that the centralised monitoring station to which all TMP nodes are connected will be used to generate performance statistics and displays that can be used to analyse faults in the system.

Unlike the full hardware solution of Tsai, TMP could be considered a hybrid of both hardware and software. This is definitely the case with the specification of event triggers used by each TMP node to control the acquisition of events and data. Event triggers are specified by load-store instructions placed in either the kernel or the application. An event is specified by loading an address with a particular value. Different events are specified by loading a different value. Such a value might signify a process event such as creation or destruction. The use of event triggers causes minimal interference, Haban quotes that in a typically instrumented monitoring session an average of 700 events per second were generated at each node. Actual development node overhead caused by the TMP node is 0.1 percent.

The TMP system was used to develop and observe the behaviour of applications for the INCremental Architecture for distributed Systems (INCAS) environment. Although the applications have not been described in detail it seems reasonable that the TMP system could be used for the development of distributed embedded applications under the INCAS environment.

The support of concurrency by the debugger developed on top of TMP is based on the event trigger specifications—that is, an event breakpoint based model. The debugging assertions are predicates based on event triggers using temporal concepts to capture relations, such as *happened-before*, *simultaneous* and *not-occurred.* Predicates may span more than one machine. Logical clocks on each TMP node evaluate the predicate and co-operate with each other until the predicate becomes true. The centralised monitoring station is only notified when the system is halted. The synchronisation of events and data from TMP nodes arriving at the centralised monitoring station uses a happened-before relationship to provide a partial ordering of events as well as using clock synchronisation mechanisms to guarantee that each TMP node reports events from the same time period.

The TMP debugger and monitoring system do not support heterogeneous domains. At the time of implementation, only Motorola based processors were supported. Much like Tsai's monitoring system, the use of hardware means that retargetting the debugger can only occur if the TMP node is redesigned. The same arguments and observations for Tsai apply for TMP as well.

Support for configuration management of development hardware as it changes over time is not explicitly handled though it is possible that events could be specified for this purpose. The level of abstraction that TMP works at is the event level rather than the resource or state level.

**Useful Contributions**

*The TMP system provides excellent support for concurrency allowing the user to specify true distributed event breakpoints.*

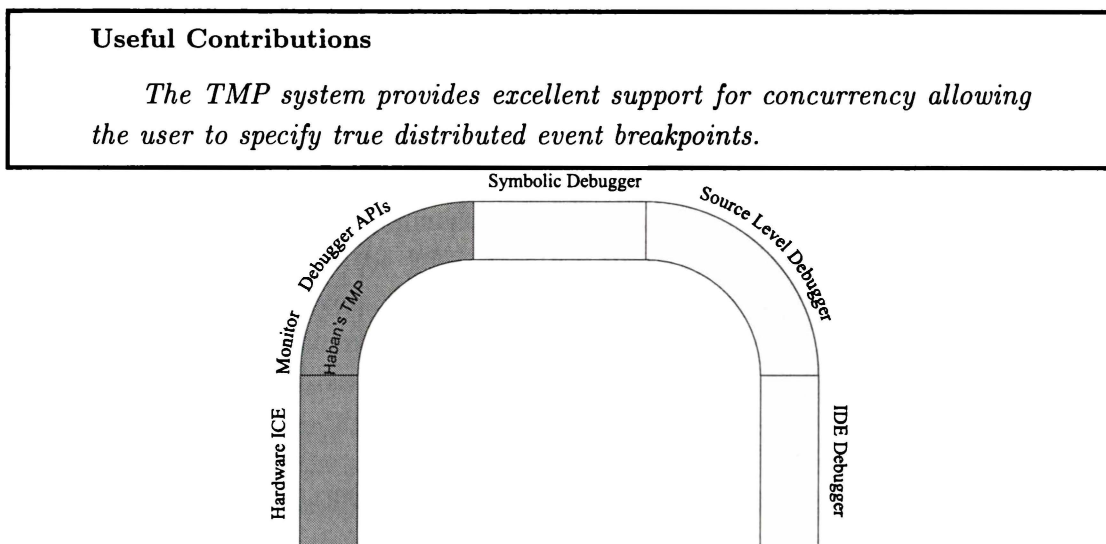



Fig. 2.4 *Haban's TMP monitoring and debugging system*

## The Topaz Teledebugger

The Topaz TeleDebug (TTD) facility developed at DEC Systems Research Centre by Redell [Red89] provides a remote debugging capability for supporting software development using the Modula-2+ language (an extended version of Modula-2).

TTD was used in the development of the TaosSystem environment which was an implementation of the Topaz operating system for the Firefly multiprocessor architecture. The Firefly hardware architecture is tightly coupled with several MicroVax-II processors connected to a common shared memory.

Even when debugging local processes, TTD is a remote debugger. All debugging is accomplished by the sending of messages between the *Loupe (the debug client/debug agent)* and *TTD-server.* The messaging protocol is defined by a remote procedure call (RPC) interface supporting the typical debug operations such as controlling execution and examining/modifying memory. The messaging protocol is implemented on top of UDP/IP datagrams.

Because Topaz itself provides a rich development environment in terms of tools and services, it is probably not used in a strictly embedded context. Applications debugged with TTD included operating system code, windowing systems and network servers.

The probe effect is not explicitly minimised by TTD it is essentially ignored. Evidence presented by Redell indicates that for the applications debugged, this was sufficient in practice. Neither is concurrency between application threads on a target or across multiple targets addressed. State based breakpoints are used with no explicit co-ordination to stop all threads on a target or other targets. TTD remote debugging is strictly a host-target one-one mapping.

Although there is nothing to prevent the implementation of the TTD protocol on other targets this has not been done. Due to the use of Topaz as a service environment that is abstracted somewhat from the hardware, no mechanisms for handling hardware or software reconfiguration while the Loupe-TTD server connection is active are present.

**Useful Contributions**

*The most useful lesson learnt during TTD development was that the debug server is best implemented at the lowest possible layer to increase availability and reliability. A uniformly applicable RPC facility is the key to enabling this approach. Limiting a single target to having just one active session with a debug client simplifies the TTD protocol.*

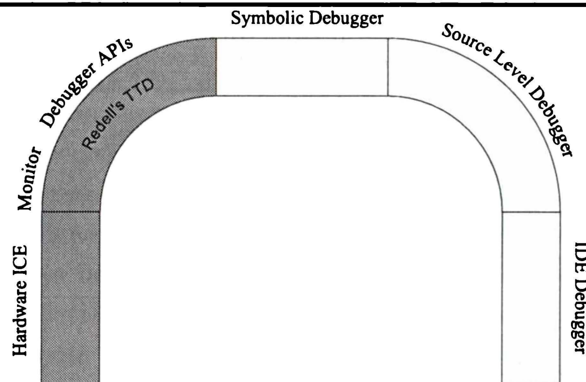



Fig. 2.5 *Redell's Topaz TeleDebugger*

## The Universal Debugging Interface (UDI)

During the period of 1992-1995, an industry endorsed API known as the Universal Debugging Interface (UDI) emerged. The UDI specification was developed by AMD [Den92] for the purpose of providing a standard API for debugging tools that are used to develop embedded systems hardware. The result of this has been a set of procedures and structures that define a debugger-monitor interface. One of the major aims of UDI is the development of systems in a cross development environment in which many embedded systems are developed.

The UDI specification separates a development system into two components, the debugger front end (DFE) and the target interface process (TIP). These equate to the debug client and monitor components described in Chapter 1 with the DFE existing as part of a host development environment that communicates with the TIP on the target system. The communications medium may be a serial line or an IP socket. The TIP could also be just another process on the host system that emulates the intended hardware environment. In a manner similar to TTD, UDI provides an RPC interface to the development target.

A TIP uses a *UDIResource* to implement access methods to manipulate resources and structures. To use all facilities provided by a TIP, the DFE must be aware of that TIP's capabilities. All

UDIResources are modelled as a space and offset. This access model is used to manipulate memory space and register contexts. Three methods are used to manipulate a UDIResource: a read memory block, write memory block and copy memory block operation. Unfortunately, due to the way in which the UDIResource is modelled, the UDI breakpoints model cannot be implemented in terms of a UDIResource.

The UDI specification does not describe methods used to minimise the probe effect although it would depend on an implementation of the specification as to whether the cost of a UDI debug server was too high. Implemented as software there will be some cost.

UDI concurrency support for distributed systems is limited. While the specification describes DFE-TIP configurations of one-one, one-many, many-one and many-many, only local breakpoints can be specified for a TIP. Specification of global or group breakpoints is not possible. Capturing global state would be complex because TIPs cannot co-ordinate with other TIPs to determine a global halt condition. The UDI specification only permits the debugging of programs on multiple TIPs where there is no communication between targets—hardly a useful distributed environment. Much like TTD, UDI specifies debuggers that are remote debuggers with little in the way of distributed application support.

Support for heterogeneous environments is limited under UDI, a DFE must be compatible with a TIP to which it connects. If the DFE does not provide facilities that allow access to all the resources that the TIP can access, then debugging is limited by the configuration understood by the debugger. The general assumption that seems to be implied by the *UDI* specification is that the DFE and TIP should be mirrors of each other; hence the debug utility already knows the target configuration upon which the TIP resides. This is probably why the support for getting target configuration is optional. In the practical sense, a debugger that conforms to the *UDI* specification can most certainly communicate with a TIP that conforms to *UDI*. Different vendors can in practice work independently of each other, with one supplying the TIP software and the other vendor the DFE. The DFE process, however, can only sensibly debug one type of TIP process for a given target configuration. A DFE in its most basic form is *statically retargettable*, that is, it requires porting whenever the TIP changes.

Basic support for querying target configuration is provided by UDI. It is unknown whether this is used to assist in the development of DFEs that can inter-operate with TIPs although it could be used. While the UDIResource model could be useful for dynamic configuration of hardware, it does not appear to have been used in this capacity.

**Useful Contributions**

*The UDIResource is a useful albeit very simple abstraction for modelling hardware resources. UDI does not mandate a particular communications medium over which debugging takes place.*

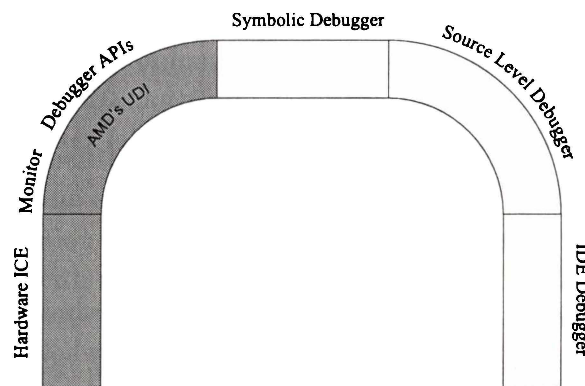



Fig. 2.6 *AMD's Universal Debugger Interface*

## Ldb

*Ldb* is a retargettable debugger design built by Ramsey and Hanson [Ram92]. It is designed to debug C programs on a variety of architectures such as the MIPS R3000, Motorola 68020, SPARC and VAX. Due to its design it can use a network to connect to faulty processes and can also do cross architecture debugging.

The ldb debugger is divided into two components, ldb itself as the source level debugger and a debug "nub" (the nub terminology comes from Redell's TTD) which is used to control the target process. The debug nub is small enough to be linked into the program-under-test executable. Ldb exchanges messages with the nub using a machine independent protocol and establishes a connection to the nub by:—

a. connecting to an existing process over the network.

b. forking the target process as a child.

c. being forked by a faulty process asking to be debugged.

An accompanying tool, *lcc*, a retargettable compiler produces code for any of the above architectures. When a target program is being debugged it does not matter what architecture ldb runs on as ldb's machine dependent code depends only on the architecture of the target program and its nub run on. Cross architecture debugging is no different to single architecture debugging. Ldb as a result can change architectures on the fly becoming a dynamically retargettable debugger.

Ldb uses *abstract memories* to model code, data and register space etc. A wire memory is used to model the connection between ldb and the nub. An abstract memory may be fetched from or stored into. Other abstract spaces may be defined based on the architecture type. The data types implemented are limited with three size of integer (one, two and four bytes) and three sizes of floating point (four, eight and ten bytes).

Ldb's main area of use has been under a Unix environment, it is not useful for hard real-time embedded systems. It assumes an underlying IP sockets and signals infrastructure to support connection between ldb and the nub. Because the nub is implemented as part of the target program there will always a level of interference. The nub ignores the probe effect.

Like UDI and TTD, concurrency support is limited to state based local breakpoints. Ldb has no defined mechanisms to debug distributed applications in a co-ordinated manner although it can connect to multiple nubs.

As described above, due to the design of ldb and the nub, the debug client is dynamically retargettable—potentially this means one debug client can be used to debug a number heterogeneous targets. In practice it is not clear that ldb is capable of this. Data representation issues are handled by the wire protocol, values are sent back to the ldb in little endian order.

Abstract memories appear to be defined statically. While ldb may be retargettable, neither it nor the nub handles dynamic configuration of hardware or software resources during run-time. When new features become available, ldb and the nub have to be rewritten to support these features, an additional set of registers for example.

> **Useful Contributions**
>
> *Abstract memories are like UDIResources and model hardware resources. Dynamic retargettability means that only one ldb is required to debug a target executable regardless of the target architecture that the program under test is executing on. This is as long as it is one of the supported targets architectures and the nub has been linked into the target executable.*

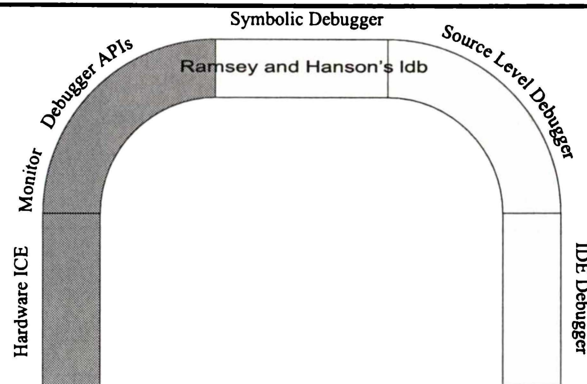



Fig. 2.7 *Ramsey and Hanson's ldb and nub*

## Pi and Blit

Pi and Blit are remote debuggers derived from the research of Cargill [Car83][Car85][Car86] at AT&T Bell Labs. Pi is a direct descendent of the Blit project, a graphics terminal with local processing capability. Pi is an enhanced version of the Blit debugger in the sense that it is written with retargettability in mind whereas Blit was only targetted toward the Blit intelligent terminal. For the remainder of this section only the Pi debugger will be considered.

The Pi debugger was designed for environments where high resolution intelligent graphics terminals were connected to minicomputers such as the VAX. The graphics terminals were similar in concept to X terminals but with the major difference of allowing a user of the terminal to run programs locally on the terminal and display output in a window based graphical user interface.

The Pi debugger was implemented as a client-server architecture using RPC. The Pi debug process executed on the host machine (the VAX) and made remote requests to a *terminal agent* that used terminal OS functions to access the program-under-test. Any graphics output such as the debug client user interface is sent to the terminal display, the human user uses the input facilities provided by the terminal to perform operations using the Pi debugger process. It is more correct to say that the Pi process is a client of the terminal agent and that user I/O directed to the Pi process makes it a server of I/O requests.

The suitability of Pi for embedded application development depends upon the target environment that Pi was ported to. In cases where Pi was ported to debug applications running on the graphics terminal such as the DMD 5620, local processing support was through a light weight kernel. This meant that real-time applications could be developed without too much underlying support. In other cases where Pi was ported to the VAX, embedded applications support is limited by the support of the underlying OS framework.

The probe effect is also ignored by Pi. Other than the multiple process concurrency there is no support for distributed concurrency. Only local breakpoints per process on one target are allowed. From what is described, it is not possible to have several Pi terminals co-ordinating a group of processes executing on different terminals concurrently.

Pi has been retargetted to platforms such as the VAX, Unix and Motorola 68000 platforms. This has been achieved using C++ to implement a class hierarchy based upon a *Core* object that defines target-independent abstractions that derived classes implement *target-dependent* (refer to Glossary—Annex J) behaviour. Unlike ldb and UDI, Pi does not have a uniform modelling mechanism for resource access. There is no mechanism for adapting the debugger to changes in configuration, the emphasis of Pi is more on tool development rather than embedded applications requiring an intimate knowledge of the hardware.

**Useful Contributions**

*Pi is an example of good design that minimises the amount of code required to be rewritten when retargetting a debugger through use of the appropriate abstractions.*

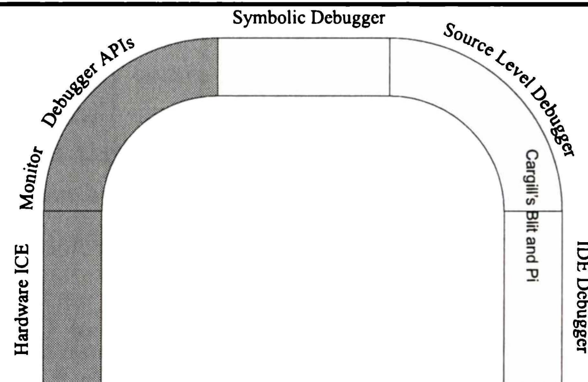



Fig. 2.8 *Cargill's Blit and Pi Remote Debuggers*

## Other Efforts—the Testbed and LPdbx

The Testbed developed by the New Zealander Burgess [Bur94] for embedded systems development is a Transputer based platform connected to a host workstation. Its main purpose was to provide an operating environment that allowed for:—

(1) Dynamic alteration of program operation.

(2) Process migration.

(3) Reliably routeing messages through an interconnection topology independent of the physical topology.

(4) Background debugging.

The approach taken by Burgess is that of the development of an embedded operating system. Its major importance to this thesis is its feature of dynamic alteration of program operation. Burgess implemented a *slot* model that allowed any program running on the Testbed to change its operation during run-time. This satisfies the configurability criteria described earlier although at a somewhat higher level.

The *LPdbx* debugger of Sorel [Sor94] is designed for loosely coupled Motorola based 680x0 processors. LPdbx has a design architecture similar to Ramsey's *ldb* but without the emphasis on ease of retargettability. Instead, the design has placed more emphasis on concurrency support,

albeit at the level provided by the Armstrong kernel that runs on each loosely coupled Armstrong multiprocessor node. LPdbx connects to monitors which control the execution of the program-under-test; examining state, manipulating breakpoints and trace facilities.

LPdbx is invoked in a similar manner to ldb, a special debug library stub is linked to the executable to be debugged and the monitor is started by the stub. The master debugger (debug utility) executed on a front end processor based on a Sun 3/60. A message monitor known as *Madcapp* is used by LPdbx to analyse timing errors. The standard monitor performs traditional state based debugging. LPdbx supports state based breakpoints but neither the master nor the monitors allow specific co-ordination of breakpoints.

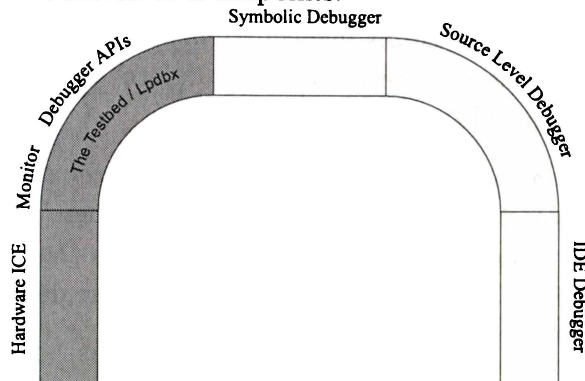



Fig. 2.9 *Classification of the Testbed and LPdbx*

## My Place

The following diagram attempts to describe the relationship between the models presented in this thesis and related work in the field. While all efforts described have a "place" that characterises fundamental architectural decisions it does not mean that this measurement is absolute for all metrics used to classify a distributed embedded debugger. For example, just because a debugger uses hardware to minimise invasiveness does not mean that it is difficult to use.

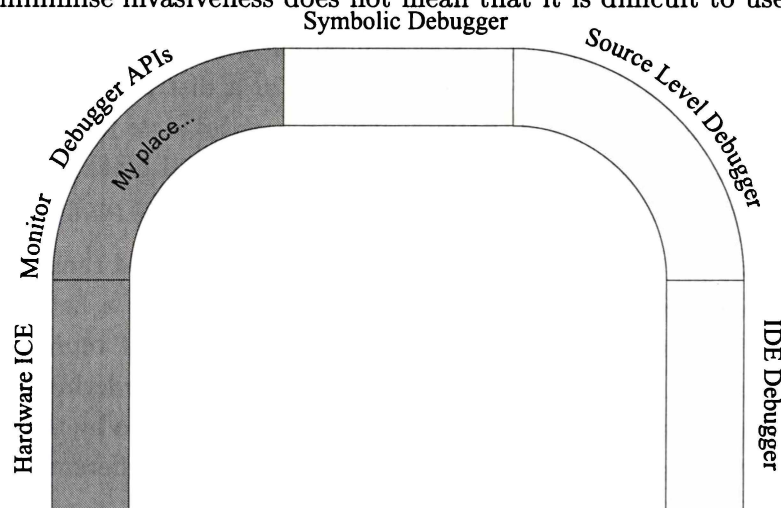



Fig. 2.10 *A Classification of Debugging Tools for Distributed Embedded Systems*

This thesis is about the design of a protocol and model for debugging loosely coupled distributed systems and of the associated tools that use this protocol and model; the monitor server functional core and functional core adaptor. The intended domain of this protocol is to allow the debugging of *distributed embedded systems* such as the field of process control used in automated manufacturing in industry. This work on what is collectively known as *remote debugging* and *retargettable debugging* builds on previous research in the field. Previous tools have exhibited great strength in some of the areas mentioned and fallen short in other areas. Many debugging tools, single processor

or distributed, have provided only very specific solutions which had little further applicability to other domains. This research demonstrates that it is possible to design models and protocols that allow the construction of *retargettable* debuggers for *distributed embedded systems.*

## Summary

The first two sections of this chapter provided further definition of the field upon which this research is based. The last section suggested a model for distributed debug tools for embedded systems. Several documented efforts were measured against this model to determine whether criteria were already satisfied. While some of the research mentioned has come very close to ideal in some areas, the same tool has been clearly inadequate in other areas. It is hoped that in reading this chapter the reader has a greater appreciation of the "place" of other work in the field as well as distinguishing where this work lies relative to other efforts.

I view the design of a *debugging architecture, debugging protocol* and *programming interface* as being of first importance. Secondly, the proof of correctness of these is essential if the results are to be usable in safety-critical systems. While the user interface to the debugger is important, it is considered to be irrelevant unless and until the above attributes are designed correctly. The design of such user interfaces is not within the scope of this research.

The debugging model must address a number of situations which will (or may) occur when debugging a distributed system:—

(1) Once a particular process on a given machine has been isolated then functionality similar to that of most modern day sequential debuggers is needed. Facilities such as tracing, breakpoints, loading of code and the examination/alteration of CPU resources such as memory/registers comprise a minimal set of capabilities that all debuggers will therefore need (McDowell [McD89]).

(2) Given that distributed computing is highly communication oriented there is a need to handle events that are the result of interactions between processes on different processors. This communication could be described as a synchronisation event. The protocol therefore needs to handle the temporal nature of processes executing in a distributed system. It also needs to provide a mechanism to allow a distributed system's *global state* to be captured in a timely manner. Thus there is a need for the notion of a group or global breakpoint in addition to the local breakpoint mechanisms that are used to debug non-concurrent programs on uniprocessors.

(3) The debugger must be capable of debugging a wide and varied range of different targets. Target configurations must not be assumed. The network may be a heterogeneous collection of target CPUs. The *protocol* must provide a common means of representing data that is represented differently on various CPUs. Problems such as byte ordering have to be handled. The idea of representing an abstract machine in an independent form by the protocol also needs to be considered. Various targets have different register sets and different ways of representing CPU registers such as program counters, frame pointers, etc.

(4) Finally, the debugger protocol must be able to provide support for the embedded systems domain. One of the tenets of embedded systems is that software and/or hardware may change dynamically at run-time. The fact that hardware may change *during* the lifetime of an embedded system is important because an embedded system is often viewed as an on-going, never-ending process where the system reacts in response to stimuli in its environment. Such activities may not allow the system to be switched off for the expansion, repair or revision of hardware and software modules. Debugging protocols must therefore also support the concepts of dynamic configuration of hardware modules in such systems.

The main body of this thesis therefore addresses the design of distributed embedded system debugging protocols, their finite state machines and correctness.

# Chapter 3

# Building Blocks to an Architecture

One of the major aims in the design of the debugging protocol is to minimise intrusiveness of the program-under-test. Absolute non-invasiveness is impossible with the protocols and models developed in this thesis if the implementation is purely software bound. From a practical standpoint, all that can be done is to *minimise* invasiveness. This means keeping the debug protocol "lightweight" by keeping it simple with the aim of minimising servicing overheads. One of the methods that can be used to achieve this aim is the use of *local* processing to minimise the amount of interaction between debug agent and debug server.

One of the defining attributes of a tightly coupled parallel computer is that generally the processing elements are of the same type. The architecture is regular and homogeneous, every system is configured and performs the same, regardless of where in the cluster. Distributed systems cannot guarantee this sort of homogeneity since, typically, such systems can exist in many domains over a potentially very wide geographical area. While distributed embedded systems may have smaller geographical ranges, each processing element may be optimally chosen for a component of the distributed system's job. This may mean that a variety of different processing units may exist. The debugging protocols and tools of this research must consider this heterogeneity. The nature of heterogeneity exists at two levels, the data level and the machine level. Data level heterogeneity is about how fundamental data types are represented and addressed by a machine—how are quantities such as words and longwords stored? How are floating point numbers and characters represented? The debugging protocol must be designed so that a uniform method of data transfer takes place. Machine level heterogeneity is somewhat more difficult in scope; how does the protocol handle entirely different CPU architectures with their differing register sets in a machine transparent manner? The debug protocol must somehow handle the notion of an abstract machine architecture which can then be applied to a "real" target.

Embedded systems programs are different to other applications such as word processing and image processing. Embedded systems activities are an on-going process. Process control cannot conveniently stop while the surrounding external process continues. What happens to the embedded process when, for example, hardware and/or software has to be replaced? For systems that provide for the insertion or removal of hardware while the system is switched-on, dynamic configuration is a solution to avoiding the problems of disrupting system behaviour and achieving some sort of fault tolerance. Debugging protocols must be able to handle the changes that occur in machine *namespace* when changes to hardware occur. How does a debugging session between debug agent and monitor server behave when peripheral hardware gets added or removed from the system? Not only should the system software be able to handle dynamic configuration, but the debug protocol should be robust enough that debugging can always take place.

Distributed systems typically execute code that runs asynchronously with respect to each processing element in the system. Debugging distributed systems is, as stated previously, debugging "programs in the large". The sequential debugging concepts are no longer sufficient for the job.

Further extension to the realm of distributed concurrent programming is required. The debugging protocol needs to be able to control the interacting processes of a distributed system. This means the ability to control clusters of processes with global breakpoints and global continuation.

I agree with the observations of McDowell [McD89] with respect to requirements of a good uniprocessor debugging tool. In addition to this, protocols for debugging distributed embedded systems should, it is suggested, exhibit the following additional attributes:—

a. *Uniform resource access*—this aids in simplifying the protocol with the consequential effect of potentially simplifying the implementation of both debugger and monitor software. A further side-effect of this attribute is the possibility of *extensibility.*

b. *Support for heterogeneous targets*—the protocol should not bias itself toward a particular target architecture. Packet protocol should be designed in such a way that all architectures are able to be accessed for the purpose of debugging.

c. *Support for concurrency*—a view of local state is no longer sufficient. Global state is required for the debugging of distributed systems. Support for concurrent debugging should allow for behaviour to be expressed at not only the instruction level, but also at the level of individual tasks and groups of tasks over a set of machines.

The development of distributed embedded applications requires tools that recognise that distributed programming environments are different from traditional uniprocessor applications. Much research has considered the issues of how to provide the best environment to organise the process of debugging the system under test. Other research has yielded tools that were either highly specific to a particular programming architecture or insufficiently capable to handle the demands of distributed embedded systems.

Distributed applications therefore require the development of debugging tools that address the following issues:—

a. The client-server approach to debugging. The problems of multiple machine domains.

b. The need to cope with dynamically reconfigurable systems.

c. The need for a consistent, well-structured means of accessing resources such as target hardware and breakpoints.

d. The need for local and "task" level breakpoints over one or more targets.

This chapter defines the approach taken to address the above issues considering approaches to the design of the proposed model as well as defining elements of the proposed model.

## General Issues of Protocol Design

Halsall [Hal88] defines a *protocol* in the data communications context to be:—

> *"A set of rules formulated to control the exchange of data between two communicating parties."*

For *meaningful* data exchange to take place, both parties must *intend* to communicate. For example, if a teacher lectures and none of the students decide to listen, no communication has taken place. Meaningful exchange of data can only take place when both students and teacher are listening to each other. Protocols organise the dialogue. For example, the student may not ask a further question until the teacher has finished answering the question asked previously.

The concept of the party that intends to initiate communication as well as the party that responds to the initiator's requests also has an important bearing on the nature of communication. The term commonly used to describe this is the *client-server architecture.*

## Preliminary Definitions

Debuggers that exist on single processor machines are considerably simpler conceptually than debuggers that have to work across distributed environments. The reasons behind this are:—

a. The need for multiple debugging agents when debugging a processing system whose tasks may be distributed over a wide geographical area.

b. The need to consider a heterogeneous debugging domain, that is, the debugging of tasks that may be executing on different processor architectures.

c. Support for concurrency and debugging applications that consist of many co-operating tasks.

An outcome of (a) is the client-server architecture. The distributed debugger is like any networked application, there is a need for a debug agent combined with a remote debug server (monitor server) that acts upon requests made by the client and returns results. Requests enable the debug agent to inquire about the current state of the remote system provided that remote system understands the request. To allow for a sensible exchange of requests between client and server, protocols need to be defined.

The outcomes of (b) and (c) which are of considerably greater significance will be discussed later in this chapter.

Before further discussion however, it is necessary to define some terms that will be used in describing the debugger architecture and protocol:—

a. *Debug Agent*—A software entity executing on a host that implements the debugging back-end protocol which allows the *Debug Client* to access and possibly modify the resources of the target system. The debug agent makes requests through some communications medium to the *Monitor Server* on the target system.

b. *Host*—Any system upon which the debug agent and debug client/utility can execute.

c. *Target*—A computer system upon which the monitor server and program under test can execute.

d. *Debug Client/Utility*—A software entity that allows a user to access resources via the debug agent software. Such software provides a user interface to manipulate target resources via the debug agent.

e. *Monitor Server*—A software entity that executes on a target system as an *actor*. When requests are received from the debug agent, the monitor services these and sends a reply back to the debug agent (and usually, therefore, subsequently to the debug client).

f. *Monitor*—A software entity that executes on a target, providing a uniform interface that allows access to both target hardware and monitor software resources. The monitor server uses this interface to serve debug agent requests.

g. *Program-under-test (PUT)*—A software entity all or part of which exists on a target system. This entity represents the software developer's application that executes on particular embedded systems target(s).

NOTE For the intents of this and subsequent discussion, the terms *Monitor* and *Monitor Server* will be referred to as the *Monitor*.

The following principle underlies all other principles presented later in this thesis. These have been separated into boxes because of their importance to the discussion at various points.

Figure 3.1 shows the fundamental debugging architecture which these definitions have described.

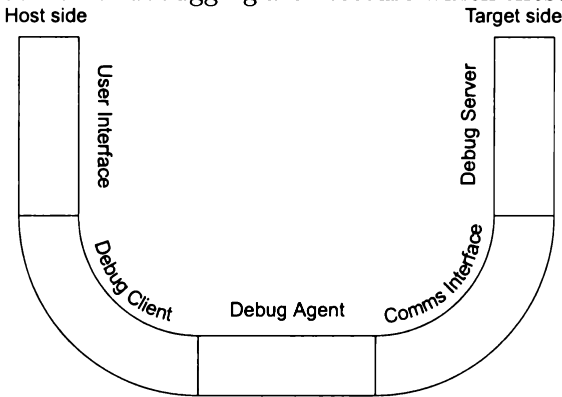



Fig. 3.1 *Debugger Architecture*

> **Principle of Decidability**
>
> *It is ultimately the human user that makes decisions regarding the correctness or otherwise of data gathered by the functioning of the debug agent, monitor and message protocol. Neither the debug agent nor the monitor have the ability to decide how to handle errors arising from the execution of the program-under-test or the user defined handlers of the monitor itself.*

Many development systems available on personal computers and workstations provide a remote debugging facility whereby a user can debug a program running on a remote machine from another machine of the *same machine architecture and operating system.* However, such debuggers assume an environment that may not exist in a development system due to the absence of system software (a bare system) or through a lack of resources (inability to support system software).

Ideally the monitor should provide "minimal" operating system support to allow debugging to be accomplished in addition to being able to bootstrap application code. Placing the monitor software at the lowest level also achieves the goal of being able to debug *all* software as discussed in the previous chapter.

> **Principle of Applicability**
>
> *The protocols and models designed are primarily intended for the development of distributed embedded systems software. However, there is nothing to prevent the monitor server being a machine that is just another host system with operating system facilities.*

In stating this, however, it should be noted that the ability of the debugger to debug software at levels below that of services used by the monitor server is compromised. For example, if the debugger uses the same code being debugged then nothing reliable can be determined from either the debugger or the PUT because of this dependency. This restricts the ability of the debugger to that of desktop operating system debuggers such as *Dbxtool* and *gdb* that do not allow kernel level code to be debugged.

Debug client and debug agent have fewer constraints due to the fact that they generally exist on systems that probably have considerably more system software support and hardware resources than the monitor software on a target. This does not mean that they should be implemented

inefficiently. Given the resources, the debug agent software should take as much advantage of host system facilities as possible.

### Client-Server Topology

A client-server model only describes part of the overall debugger architecture. Debugging distributed systems requires the ability to debug an application whose component tasks may exist over a wide geographical area.

The debugger user interface may be implemented in such a manner that multiple sessions between the debug agent and monitor servers are possible. This is to be contrasted with the user interaction required for debugging a program on a non-distributed system where there is a one to one relationship between the debugger and the program. For example, a distributed embedded systems application will generally exist over more than one target. For a debugger to control the behaviour and maintain a centralised view of such a system of co-operating tasks, the debugger must be able to support multiple sessions to an arbitrary number of target machines.

Halsall [Hal88] uses the term session to describe a *dialogue* between communicating parties—in this case, the debug agent and the monitor server that is running on target hardware. For each debug agent there exists a session dialogue to a particular monitor server. Is there any sense in being able to have a number of clients connected to one monitor server? There are two aspects that need to be considered:—

(1) The need to be able to always *view* but not necessarily modify the resources available on the target

(2) The need to arbitrate write access between sessions on the monitor.

The two aspects, described above, are related. The work of Redell [Red89] on the Topaz TeleDebugger considered the high availability of a debugging service applying the motto, "You can always debug". The Topaz TeleDebugger allowed multiple sessions to be connected to one target. A debugger should be able to arbitrarily connect to a target which may currently be monitored or debugged by another session. A debugging session between a debug agent and monitor always provides *access* to all target resources. The term *access* is used in this context to mean at least *read* access to target resources. The multiple session model has the benefit of *multiple views*, each session could represent a view of program behaviour. The concept of multiple views is complicated by the decision of whether each session has read and write access to the resources of that target machine. Such access needs arbitration which leads to the familiar problem of *multiple readers and one exclusive writer*. If the monitor had a number of sessions with debug agents, only *one* session is allowed read and write access, the other sessions should only be allowed to read access to resources to avoid conflicts that would occur if sessions were allowed access in an unco-ordinated manner. Having one writer results in trivial arbitration issues. There are no problems of mutual exclusion because the writer session is exclusive.

### Client-Server Design Constraints

The problems with supporting multiple sessions on the monitor is that session management becomes more complicated. This becomes readily apparent when considering dynamic configuring systems and the problems of connection loss when either debug agent or monitor "crash". There are two approaches to this problem; either provide for this extra complexity and complicate the session state machine on the monitor or ignore the complexities and leave the debug agent to handle these situations when they happen.

The protocol model only supports the concept of a single session between debug agents and one monitor server. The main reasons for this are:—

(1) Multiple sessions are difficult to manage. The monitor has to track the originator's requests and respond to the originator in kind. To support multiple sessions requires the use of history mechanisms that associate the originator with the request.

(2) Multiple sessions complicate the issues of handling *dynamic configuration* and *break event handling.*

It can be argued that multiple sessions, while useful, do not occur very often anyway. Read-only sessions could be treated as "pseudo-sessions" managed by the debug client that do not have the same facilities available to them as the "real" read-write session. The problem then reduces to functionality to be managed by the debug client and not a part of the protocol model.

**Principle of Session Multiplicity**

*Only one session is allowed at one time between a monitor and debug agent.*

At this point it is worth re-examining Figure 3.2 (originally presented in as Figure 2.1 in Chapter 2) in more detail. It is important to note that while the diagram may indicate that a host and target system (and consequently debug agent and monitor) are directly connected to each other, the actual physical configuration of communication may not necessarily be point to point. Ultimately the physical topology is determined by the communications hardware that is provided by both host and target machines. For example, Ethernet is a bus architecture while Token Ring uses a ring topology to connect machines together.

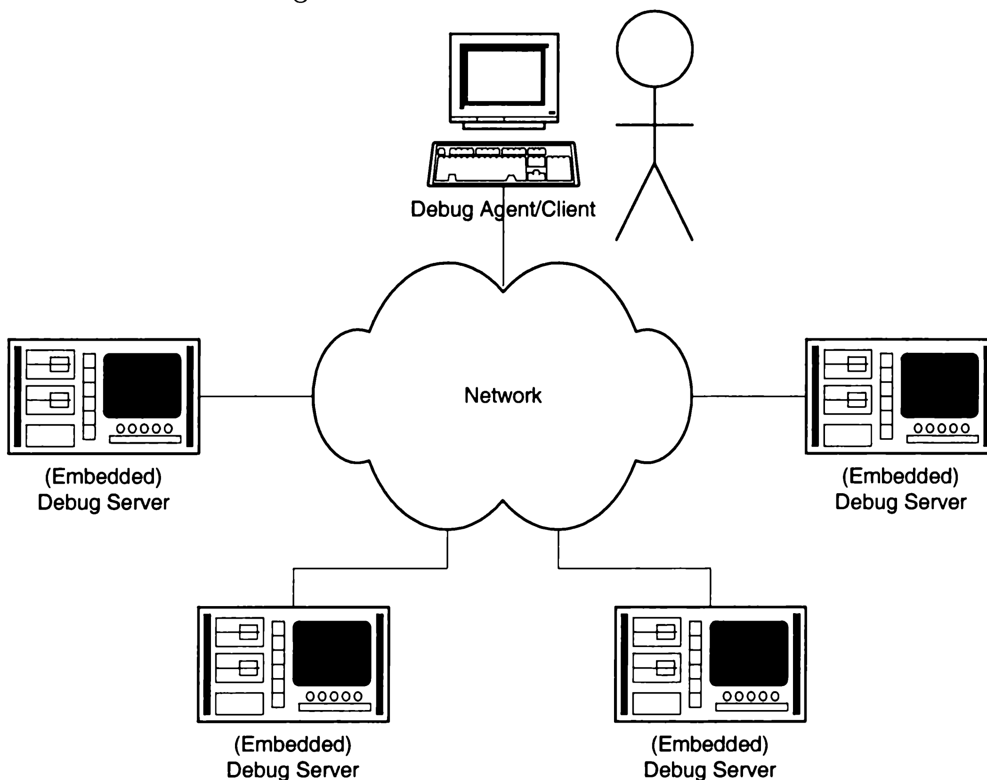



Fig. 3.2 *Debugger-Monitor Session Architecture*

In designing any communications protocol the decisions that typically have to be made with regard to servers are:—

(1) Whether they are connectionless or connection-oriented.

(2) Whether the servers maintain state or are stateless.

A connection-oriented interaction is used to describe the reliability of the connection. A reliable message transport ensures the verification of message contents, the preservation of packet ordering through sequence numbering, the re-transmission of corrupt messages and the flow control of messages.

Connection-less interaction presumes, at the cost of reliability, a "best effort delivery" approach. In a connection-less service, a message sent may be lost, duplicated or delivered out of order with respect to other packets. Protocols must be robustly designed to recover when such errors occur. A connection-oriented transport on the other hand requires more complicated software to maintain connection state to guarantee connection reliability.

The choice of communication interaction mode has consequences for the design of debugging protocols for embedded systems. Such systems may not have the resources to implement complex algorithms that allow reliable transport of data. The design of a sound debugging protocol is constrained by the ever wise "Keep it simple, stupid" maxim and the need to consider that target resources may either have their use restrained or completely prohibited. A good example of such a resource is that of a timer. Timer hardware is often used to provide a time-out mechanism to permit re-transmission of messages. The usage of such resources may make sense when they are readily available and arbitrated fairly on host systems with operating systems support. However, fair resource usage cannot be guaranteed with target systems whose application programs may use *any* resource at their disposal to implement the embedded system software. To avoid such problems occurring:—

> **Principle of Service Interaction**
>
> *The debug protocol is designed using a connection-less interaction for reasons of simplicity and minimal resource usage. The inherent reliability is part of the protocol design since the transport is used only as a best effort message deliverer.*

It is important to note that while the interaction style may be connection-less, the concept of a *session* is still valid. The task of maintaining a session between two communicating parties gets moved to the level of the debug agent and debug client.

The second concern is the need to maintain state for interaction purposes. The advantage in maintaining state in servers lies in efficiencies gained by the reduction in the size of messages and the consequent potential speed increases. Designing stateless servers provides the advantage of protocol reliability, if a state-dependent server loses messages, receives duplicate messages, receives messages in the wrong order or the agent machine crashes, server state may become invalid causing incorrect responses to be issued. While the overhead to maintain state within the transaction is larger, a stateless server will always be in a position to retry a transaction and carry on regardless of the problems experienced by protocol transport. Ultimately the issue of statelessness is that of whether the protocol is responsible for reliable delivery. To minimise or eliminate the need to maintain server state, protocol messages must be carefully designed so that they do not depend on previous message context—messages must be complete and unambiguous. Protocol design should have the server respond in the same manner regardless of how many times a request arrives. Mathematicians use the term *idempotency* to describe such a property.

> **Principle of State Minimisation**
>
> *The debug model should be designed to minimise the state to be maintained in such a way that individual messages are self contained entities and do not depend on the contents of previously sent messages.*

Ultimately, the monitor cannot be completely stateless, there is some state that has to be maintained, particularly with regard to the session state. However, the above principle should always be an underlying design guideline.

### The Demands of Developing Applications

Development of applications for embedded environments places demands that are different to those faced by an application developer who uses the facilities offered by an operating system framework such as Unix or Win32. None of these facilities may be present in an embedded systems development platform. In such a case, the application developer will probably need to develop facilities to manage the resources of the development platform that might be normally managed by the operating system. For example, the application developer might have to develop a physical memory manager or a kernel to manage a list of tasks.

In a development system such as that used to deploy an embedded application the following steps might be taken when developing an application:—

(1) Use the host facilities such as editors to design and develop code.

(2) Use host tools to cross compile and cross link the application to execute on the target processor's architecture.

(3) Test the application.

(4) If the application does not function as specified, further develop, build and test the application.

The third step is somewhat vague. How is the PUT tested? It is not practical to build a ROM image of the PUT and attempt to execute it on the target hardware because there is no way to debug by inspection of state or controlling of its execution. What is needed is a reduced form of bootstrapping code that can be used to test the PUT—the monitor.

The monitor provides the very basic facilities of primitive operating system executives of the past to:—

a. Download the program to be tested from the host development platform to the target platform.

b. Control the execution of the downloaded PUT. This means primitives to commence or halt the execution of the PUT.

c. Examine or alter the state of the system.

The monitor can only perform these services, any other services should be implemented by *only* the PUT. It is not the responsibility of the monitor to provide any other services. This means the PUT should always have the entire machine resources at its disposal, even those that the monitor may use such as hardware traps.

**Principle of Minimal Resource Usage—Mark I**

*The user application being developed on the target hardware should be able to assume all resources on the system are available to it, even at the loss of debugging facilities, the monitor's resources.*

The design of the debugger model and debug protocols should also reflect this need to constrain the PUT as little as possible. The design principle of state minimisation and connection-less interaction assist in the reduction of monitor complexity.

### Monitor Resources

The ideal situation in which the PUT treats the target system resources as if they were *completely* its own is not possible if monitor operation is to continue. To be able to function, the monitor requires resources to support an execution environment (workspace), along with a need to execute code from somewhere that has to be part of the target address space. The key is that monitor resource usage should be minimised as much as possible to provide the PUT being developed with as much freedom as possible while still providing support for debugging facilities.

What resources does the monitor server require? The monitor requires memory space for both data and code. In addition to this, the monitor also requires a *channel* through which communication with a debug agent may take place. The term *channel* is used in this context to define a communications medium that allows bi-directional, half or full duplex operation between two or more machines which may be targets or hosts. Connection topology could be multi-drop in the case of serial lines, bus based in the case of Ethernet or ring based in the case of a Token Ring. More than one channel, each representing a potentially different access medium may exist on either target or host machine. Any code that constitutes a *device driver* may need to rely on polling or interrupt mechanisms to effect data transfer.

**Principle of Minimal Resource Usage—Mark II**

*The debug protocol is designed to function with the least possible resources. In this manner, it is able to provide a development platform upon which bootstrapping of embedded systems software is possible.*

The monitor server cannot rely upon any lower level kernel facilities since the monitor server *is* the kernel.

In order to permit *interactive* debugging the monitor has to have exclusive use of channel resources (vectors and the registers of the device providing the channel) used by a communication channel as well as those hardware traps that provide the monitor with necessary functionality to allow program breakpoints to be set to actually provide the intended functionality—that of being able to debug the user application!

In the case of multiple channels, a different set of circumstances exists. Why not allow the monitor usage of those extra channels for debugging purposes if they exist? Some monitoring systems use a different set of channels to reduce invasiveness, applications use one set of channels, the monitoring systems use a private (in this case, a different set) of channels to allow debugging ability to be maintained. The work on TMP by Haban [Hab90] is a good example of this method of using auxiliary channels of communication. However, dynamic configuration complicates the ability to manage and use such devices when they become available on the target—this problem will be addressed later in this chapter. To state it simply for the present discussion, the monitor

cannot assume that it *always* has a set of channels that it can use by default. The monitor could have one channel, or perhaps several channels, which could be used to communicate with the debug agent.

> **Principle of Multiple Channel Support**
>
> *Under direction by a debug agent, the monitor, with appropriate protocol support, should be able to take advantage of other communication channels if they exist so that the monitor's ability to debug is maintained despite the user application's use of the "default" channel normally used by the monitor.*

The previous paragraph described the situation where more than one channel could exist. Why should the monitor software even consider this useful? The answer lies in considering when a PUT might actually *require* the channel normally used by the monitor for its communication with the debug agent. The PUT could be using that channel as part of some embedded application under test (eg., for remote data acquisition). The situation that needs to be considered is how can debugging take place when a PUT acquires the channel used by the monitor for debug agent-monitor communication? The simplest case to consider is where there is only *one* channel available for use on the target system. If this is the case then all hope of reasonable debugging while the PUT is executing is lost.

Often, communications protocols involve some notion of a time-out mechanism so that re-transmission of a message can occur if some specified time has elapsed and no reply has been received. Protocol software that is part of the monitor can neither assume the presence of nor use any timing hardware to manage message re-transmission. It is the responsibility of the debug agent on the host to manage such situations since it is the "master" of the session connection to the monitor.

## Relationship between User Application and Monitor

The monitor exists for the primary purpose of providing a debugging service for the PUT on behalf of the debug client via the debug agent. It also exists as the *only* means of controlling the execution of the PUT.

The actual flow of control could be likened to that of a co-routine. The concept of co-routines exist in languages such as Modula-2 and is generally described as a method whereby two or more separate tasks schedule *themselves* by means of *explicit transfer* to each other. In this case the monitor and embedded application are co-routines of each other. When the debug agent sends a request to the monitor to start the execution of the PUT, the monitor transfers control to the PUT which executes until it either terminates successfully or some program or monitor event occurs which causes a transfer back to the monitor co-routine. It is up to the embedded application to provide its own multitasking kernel facilities when required since the monitor provides *no* multitasking executive. This reinforces earlier statements that:—

> **Principle of Minimal Working-set Functionality**
>
> *The monitor provides only the facilities necessary to develop distributed applications in a distributed embedded environment, ie., application loading, target resource manipulation and control of execution for the purpose of debugging.*

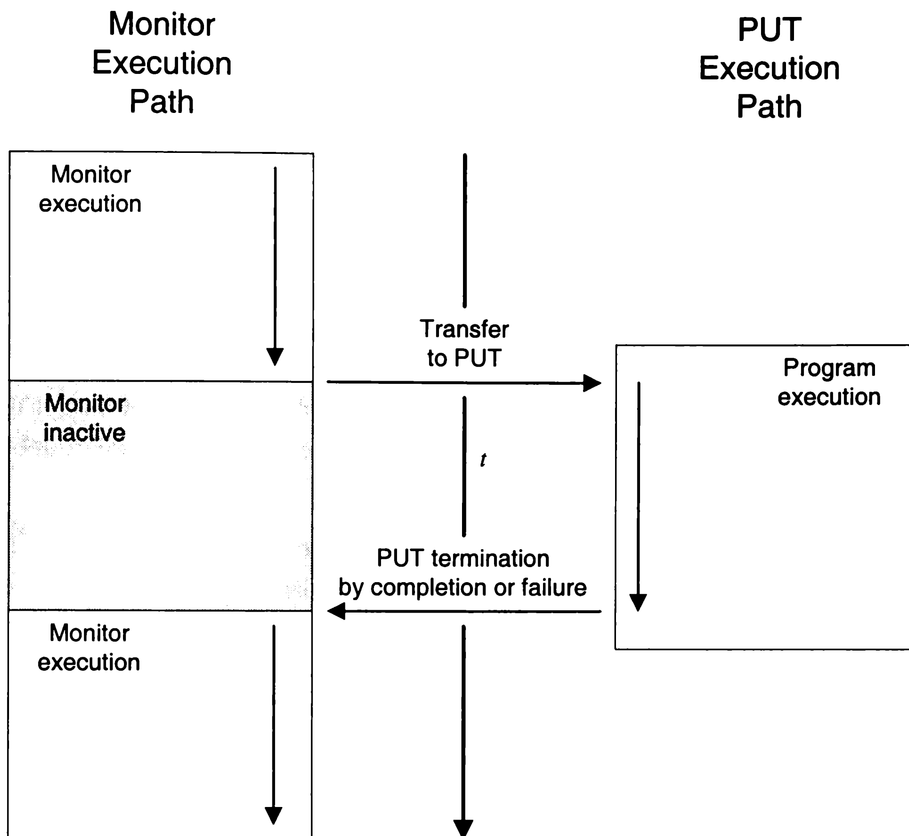



Fig. 3.3 *Program-under-test—Monitor Software Relationship*

## Dynamic Configuration

The embedded systems environment is considerably more demanding than that of desktop computing for reasons outlined in Chapter 2. The embedded applications domain has to consider not only physical constraints, package size, power usage and robust construction but also various system requirements that reflect the nature of the application and its interaction with the surroundings. Consider the following extended example:—

*An unmanned surface vehicle for planetary exploration. Such vehicles, in addition to their physical requirements of ruggedness to cope with extremes of the surface need considerable computing power to provide for the processing of data acquired by the probe's sensors. These sensors are not only used to gather information potentially useful to interested parties but also to provide the surface rover with "senses" that allow it to make decisions about where it may safely move. As rugged as the rover is, it is a likely chance that stray radiation could affect the delicate microprocessors that provide the "brain" of the rover. Since the surface vehicle is quite a distance from anywhere that can actually repair the electronic componentry, there is a need for providing sufficient redundancy in the form of extra components such as boards, microprocessors so that when the worst happens the machine might be able to repair itself. When the repairs are effected then the desired result would be a correctly functioning rover that satisfies the needs of scientists until some other minor disaster occurs.*

As extreme as this example might be, a scenario such as this is known as *dynamic configuration.* The term is used here to describe a situation where hardware may be added to or removed from a system during the execution of the application. This is in contrast to computer systems where the addition or removal of computer hardware requires the removal of power from that system.

How does dynamic configuration affect the monitoring and debugging of distributed embedded systems? It is the author's opinion that the debugger protocol must be able to handle such situations as they occur. Typical debuggers used under operating systems environments such as Unix restrict the types of software that can be debugged. This environment in both a hardware and software sense, is a static and abstract virtual machine.

Such assumptions cannot be made about embedded environments as the embedded application being debugged may have to handle changes in hardware as well as software during execution. Changes in configuration not only have to be somehow detected by the monitor, they must also be reflected in the *debug agent's* view of the target system to which it is connected. Thus as the target system's hardware "evolves", both the monitor and debug agent's view of the target changes. This target system evolution may even take place *during* a debugging session between debug agent and monitor.

Does the monitor need to keep track of dynamic configuration? If the PUT relies upon dynamic configuration to perform its task, the monitor should also be able to account for changes in configuration. To allow the monitor to access and modify resources that have been dynamically configured requires that the monitor be notified when the hardware is dynamically configured. Once the monitor has recorded the fact that hardware has been inserted (or removed), the debug agent at the host can be notified that such resources are now a part of target configuration and can now be modified. In the case of removal, the debug agent can "forget" that this area ever existed and leave that area of memory map alone. Debug agent access to target resources becomes more secure because only resources known to it are legitimately accessible, all other resource access is prevented due to the debug agent not having any knowledge of its existence.

The concept of hardware resource configuration is best supported by a combination of hardware and firmware. Desktop systems already exist (eg., so called Plug and Play buses such as Intel's PCI) where *static* configuration of cards on a backplane are configured automatically upon system start-up. What is more rare however is the ability to allow what has been termed "hot insertion" of hardware while the system is powered-on. Ideally, configuration of hardware should take place dynamically with card characteristics such as size, base address and vendor information being used to map the card into the target address space. To support such functionality it is desirable (but not mandatory) that the hardware supports some form of event such as an interrupt that signals the occurrence of hardware being added or removed.

### Information Requirements of Dynamic Configuration

What kind of configuration information should be sent as part of the notification between debug agent and monitor server? There are three approaches that take a monitor server-centric, debug agent-centric or hybrid view:—

(1) The monitor server completely supplies the configuration information to the debug agent. If, at session establishment, the debug agent considers the monitor server and the resources it manages to be a complete black box and the debug agent itself is "dumb" then at session establishment the debug agent gets a *complete* description of the resources of the target. Such a description might include a memory map, the format of all registers down to the bit level and a description of the data types supported by the target architecture. The major disadvantage is that such an approach does not reflect the practicalities of a real embedded systems environment. Normally, cross-development tools are used. Development occurs at a systems engineering level which means quite a lot of configuration information is already known. This method also wastes considerable space in describing every resource of the target completely.

(2) The debug agent stores configurations of several different "types" of target. The monitor server supplies minimal configuration information to identify a matching target description on the host machine. The monitor still has to provide a means of accessing and modifying resources that make up the target configuration. All the resource's semantic meaning such as, "which register exists at location X?" now is wholly based on the host system with some form of translation to something that the target can understand. This approach is not well suited to dynamic configuration because such events potentially change the type of configuration to something different that might not be recognised by the client.

(3) A hybrid scheme that contains elements of the above two approaches. The monitor on the target keeps *descriptors* of target resources in a way that abstracts all machine dependencies. For each resource there are routines that allow access and modification to be done to that resource. On the host side, descriptors that are transmitted at configuration time are used to look up "symbol tables" that describe the resource in a high-level manner that is suitable for human interpretation. For example, such symbol tables might map a certain value from a particular address to represent a hardware register of some interface chip.

The third approach was chosen because of its ability to handle dynamic configuration of target resources easily. This approach was developed into the abstract modelling entity known as the *namespace.*

When new hardware is added to a target system, the map of available and accessible resources becomes larger, when hardware is removed the number of accessible resources becomes smaller. How the objects are accessed and structured depends on how the hardware is structured.

## Namespace Architecture

> **Definition—Namespace**
>
> *A set of names to action mappings that model a resource. Each of the actions performs an operation particular to that resource. The namespace model allows all monitor and target system resources to be uniformly accessed, modified, created and destroyed.*

From the very beginning of this research, one of the major aims has been to design a retargettable debugger able to debug heterogeneous distributed embedded applications. The term *retargettable* is used here to describe debugging tools that allow easy cross-architecture debugging by careful, well abstracted design of machine dependencies.

a. Retargetting might be static in the sense that, in order to allow cross-architecture debugging, some code modules that represent architectural dependencies might have to be modified and the source recompiled.

b. Dynamic retargetting means that the debugger can handle cross-architecture debugging without any need for recompilation of source text.

The ability to *dynamically* retarget on a per session level without having to recompile the monitor or, more importantly, the debug agent source code is very important. The concept of dynamic configuration of target hardware makes this problem more difficult. Current debugging tools are designed for static retargettability at best. In a heterogeneous embedded environment this is not sufficient.

The key attractions of the model are that it provides:—

a. A solution to the dynamic configuration problem.

b. Simplified monitor design. The monitor is no longer a monolithic software entity but a highly extensible software object.

c. A uniform basis for protocol design.

## Monitor vis-a-vis Namespace

The namespace is an abstraction of a hardware or software resource. Names within a namespace are acted upon by handlers and results are returned based on those actions. The key idea with the namespace abstraction is that a resource, not the monitor, provides methods to access or modify itself. This may sound familiar to those practitioners of object-oriented programming although in this case such a model was *not* chosen for this reason. The choice was made due to the model's ability to solve problems of dynamic configuration. The object-oriented nature of the model came as a by-product.

In the case of a hardware resource, a namespace might represent the enumerable range of "visible" hardware that can be accessed and manipulated in some resource specific manner—program memory space could be modelled quite naturally as offsets from a base address, an offset and size of data at that offset.

The advantage of the namespace model is simplicity—the design of the monitor is simple due to the fact that the management of resources is no longer locked within the monitor. The monitor only acts as a dispatcher to various namespace handlers which act on their own namespace, manipulating and accessing the namespace through their own routines. The monitor only has to provide facilities that enable the creation and destruction of namespaces as well as the addition or removal of handlers to those namespaces.

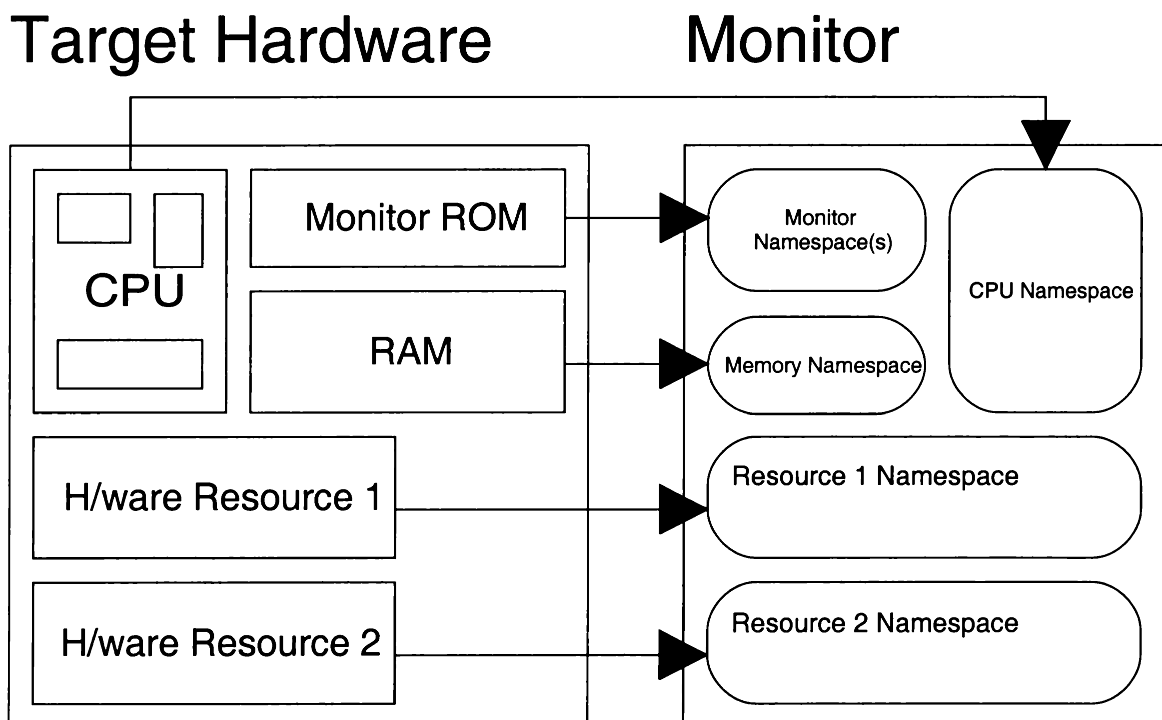



Fig. 3.4 *Monitor and Namespace Relationship*

**Principle of Loosely Bound Handlers**

*Keeping the handler code of a namespace outside that of the monitor means that dynamic reconfiguration of resources is simpler due to the fact that monitor software is loosely coupled to a particular target configuration. If hardware or software objects are dynamically created and destroyed, the monitor does not have to change its access methods to access and modify a particular piece of hardware, only the handler for that particular namespace changes. The handler does not have to be present until configuration of that device takes place. It could be downloaded at any time during a session.*

It is important to point out that the monitor server itself *never* directly accesses resources when performing debugging. *All* debugging access is done via the namespace handlers. Device dependent operations are encapsulated with each handler. It is the handler that "knows" best.

Another point to consider is that handlers that manage hardware namespaces always see physical addresses. No attempt is made to use virtual addresses. This was considered very early in the design of the debugging protocol. The handlers "see" physical addresses and the debugger does so as well. Applications being developed may however, use whatever memory management hardware is available to implement virtual address spaces.

**Principle of Physical Address Mapping**

*The monitor software must only be concerned with a physical addressing model. Memory management schemes including memory protection and virtual memory can be implemented to suit the requirements of the PUT.*

## Handler Call Interface

Handlers for a namespace operate on parameters supplied to them by messages sent by the debug agent. What the parameters are depends on the namespace involved. For example, in the case of a CPU register namespace, possible parameters could be the encoded register name. The symbol descriptors (implemented, say, as files) on the host not only enumerate and map names but also describe the handler interfaces for a given namespace.

## Namespace Modelling

NOTE Formal definition of the Namespace Model is given in Annex A using the formal specification language *VDM-SL*.

A namespace may defined as an environment (*Env*) that consists of a static work area which contains state and handlers for the namespace. The work area can be used to maintain state across handler invocations. The *map* within the environment contains name references to corresponding handlers which represent the methods by which a namespace may be manipulated. The operations performed by the handlers themselves are determined by the namespace they manipulate. For example, a namespace that represents the memory of a target machine probably contains handlers to read from and write to addresses in the memory map.

There is no limit on the number of handlers that a namespace can have as long as the workspace size is not exceeded. The advantage of having separate handlers for each resource means the monitor does not have to support all methods of access to hardware. The namespace handler "knows best"

how to access the resource which it handles. Using the example of the memory namespace again, if memory is non-contiguous and contains "holes" there is no need to write a general routine that knows the entire memory map. Namespaces allow handlers to be built that are often simpler and more reliable than general purpose handlers. When resources are dynamically configurable, the memory map changes dynamically. General purpose routines are impractical to develop, unusable and insecure in a changing environment.

> **Principle of Namespace Handler Functionality**
>
> *Handlers are responsible for providing access methods to manipulate their own namespace.*

The namespace model does not have to be used to model hardware resources, it is far more versatile than that. The handler model allows *abstraction* of any *resource*, hardware or software. The model looks and behaves much like an implementation module, as in many modern object oriented languages. The major difference is that handlers can be instantiated *dynamically* along with namespace environments. By being able to manipulate handlers dynamically the model can be applied to the solutions of dynamic configuration and distributed embedded application development.

### Resources as Objects

The namespace model is structured similar to an $n$-ary tree. The tree may grow to an arbitrary depth that reflects the resource structure of the target and the monitor namespaces. The root namespace is the top level namespace of the monitor, all other namespaces are subordinate to the root. Each namespace has a number of default handlers, the most important one being the *Find-Name* handler (used for name resolution) described in Annex A.

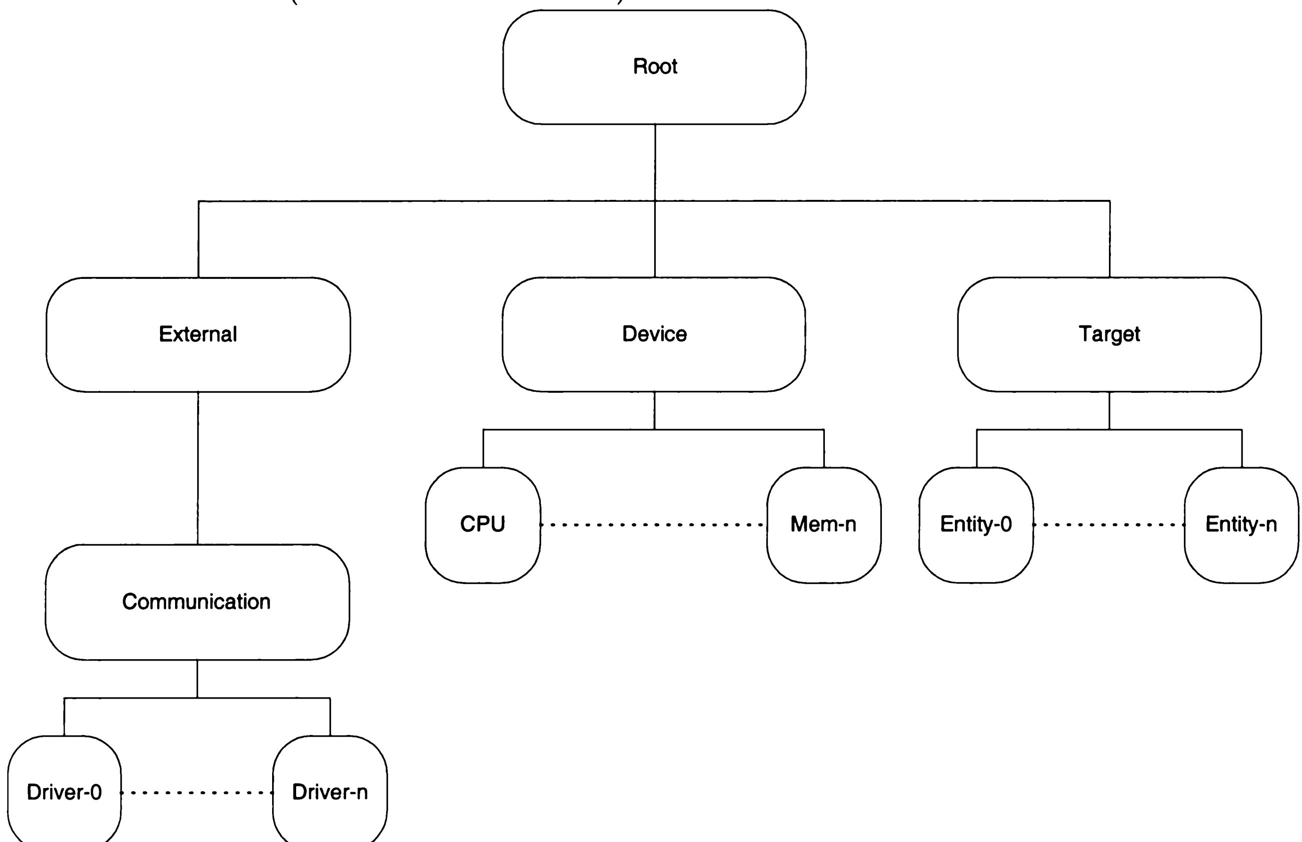



Fig. 3.5 *A simplified view of the Namespace Hierarchy*

The *Find-Name* handler is specified as a directory service—it either finds a name of the re-

quested handler and executes it or finds a name that represents another namespace and routes the namespace requests to lower parts of the hierarchy. It is a software implementation of the NAMC instruction as implemented in the Burrough's 6700 series hardware [Org73]. The *Find-Name* handler is never explicitly invoked by the current namespace, the namespace's parent is responsible. The root namespace contains handlers to create and destroy child namespaces and to create, destroy and replace handlers in the current scope. When new namespaces are generated, all these handlers are *inherited* by the child for use in generating scopes below the child.

The namespace model uses the idea of scope in a similar way to that of programming languages. Namespace handlers are invoked by providing a path to the namespace and handler name with any required arguments to the handler being supplied. The notion of a path is very similar in concept to that used in hierarchical filing systems. Requests from the debug agent consist of a path name to the monitor and required namespace within the namespace object hierarchy. Names within a namespace can either refer to other namespace environments lower down the hierarchy or to handlers within the current namespace.

**Principle of Naming Objects within Namespaces**

*A Name Msg-Element is used to denote a name in a given namespace. The name itself may be one of two things, the name of a handler or the name of a child namespace.*

A sequence of Names in a message defines a hierarchy that models the resources and entities that are present on a target. If such a sequence could be viewed as a qualified name path (Modula-2 or C++ style) then it could look like — Name · Name · Name ·...· Name · Argument.

When configuration changes do occur, only the hardware information of *the board* and a name need to be supplied to the debug agent. Any handlers associated with individual resources on the board may then be passed by the debug agent to the Monitor.

A *user* does not perceive the notion of a card that is plugged into the system, although the monitor and debug agent require that the card descriptor exist for configuration to take place. What is important is the resources and the names present on those resources—users may be oblivious as to what cards are installed on the target as long as access to the resources on those cards is possible. While this topic again borders on the debugger human-computer interface, the debugger software should only display resources that the user is interested in accessing. A card level view is only useful to the monitor and debug agent for configuration purposes. It is the debug client that maps user requests via the debug agent and sends namespace requests to the appropriate resource namespace handler on the target.

When configuration changes do occur, the monitor keeps the debug agent notified of changes in configuration. In this way, the debug agent and monitor are synchronised and see the same namespaces. The addition of a card and its resources means an addition of the card resource and an addition of resource scopes below it for the namespace hierarchy present on the monitor. An update of the additional symbols to the namespace that the debug agent sees occurs by messages being sent from the monitor to the debug agent.

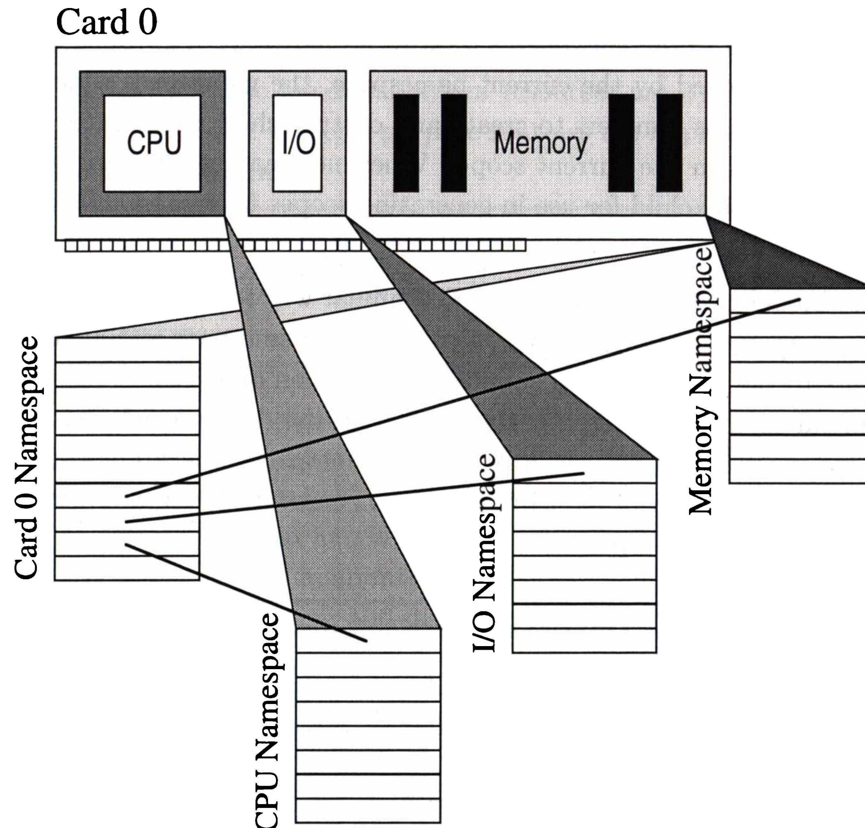



Fig. 3.6 *Resources and Handlers within Resources*

A symbol descriptor used by the debug agent should describe the resource structure and the handler *interface* but no code for that handler. The code for the handler depends on the processor configuration of the target. Handlers are always written in the native code of the target processor's configuration. Access to any resource or entity is ultimately determined by the handler implementation's reliance on how the target processor accesses the resource.

> **Principle of Handler Code must be Native**
>
> *Code to implement the handler must be that of the target processor.*

### Namespace Operations

There are six handlers provided within the namespace model for operations on namespace environments. These operations *find* names within the current namespace, *create* a namespace environment, *destroy* a namespace environment, and *add, remove* or *replace* a handler within a namespace environment. Their signatures as given in Annex A are:—

operations
*Find-Name(msg : Message, ρ : Env) res : Handler-Result*
*Add-Name(msg : Message, ρ : Env) res : Handler-Result*
*Remove-Name(msg : Message, ρ : Env) res : Handler-Result*
*Add-Handler(msg : Message, ρ : Env) res : Handler-Result*
*Remove-Handler(msg : Message, ρ : Env) res : Handler-Result*
*Replace-Handler(msg : Message, ρ : Env) res : Handler-Result*

The above core namespace model handlers are examples of what is termed an environment handler (*Env-Handler*). The purpose of an *Env-Handler* is to create, destroy, populate and depopulate

a namespace environment. Each *Env-Handler* has the same signature:—

*Handler-Result* =
*msg : Message*
*env : Env;*

*Env-Handler* =
*fun : Message* × *Env* → *Handler-Result;*

An *Env-Handler* can only manipulate the environment—the set of names and associated handlers in the environment. Its purpose is to create or destroy other namespace environments (*Env*) or manipulate a *Work-Handler.*

The purpose of the *Work-Handler* is to actually perform some requested service. A *Work-Handler* has the following signature:—

*Work-Handler* =
*fun : Argument* × *Workspace* → *Data* × *Handler-Status;*

A *Work-Handler* takes arguments and operates on a namespace's workspace (state area) yielding result data and a status that determines the success of the operation.

At the *Work-Handler* level, the argument set for a particular handler depends much on the namespace it is manipulating. The handler interface therefore cannot be defined in absolute terms since the handler interface depends on the individual resource that particular namespace models.

Namespaces are created or destroyed through changes in hardware and software configuration. For hardware, the creation (or destruction) may take place in the middle of a session dialogue between debug agent and monitor or when no session is active. In the case of program entities, the creation takes place when entities come into existence at initialisation and destruction takes place when that entity finalises. Entities represent activations and deactivations of "objects" such as threads of control *during* run-time of the PUT.

*All* namespaces are constructed by these routines either statically at monitor initialisation or dynamically when new resources or entities are added or removed. Due to the scoped nature of namespaces, the creation of a namespace requires a suitable calling environment. This means that monitor initialisation of statically configured namespaces is important.

The discussion so far has only applied the namespace model to solve the problems of dynamic configuration of hardware resources. The namespace model can also be used to model components of the monitor "kernel", particularly with respect to controlling the execution of the PUT and managing sessions between debug agent and monitor.

The only difference between namespaces that model access to hardware resources and namespaces that control monitor functionality is that monitor namespaces always exist regardless of changes to target hardware resources. The namespace model is therefore used to model *all* components of the monitor irrespective of their being a physical resource or some higher level monitor resource.

Whether the names within a given environment refer to handlers or other namespaces is irrelevant. Like the NAMC instruction, *Find-Name* has to match the message tail with a name that points to a handler so that a handler is executed within that namespace. If the name points to another namespace finder then that head of the message is absorbed and the tail is passed to the *Find-Name* of the child namespace.

> **Principle of Name Resolution**
>
> *Find-Name sees only names within its own namespace.*

NOTE The names within a namespace *must* be *unique*. Across namespaces the names can be the same because scoping rules provide a unique path.

## Downloading Handlers

So far, this chapter has mentioned the use of downloading handlers several times. What has been ignored so far is the load protocol itself. Conceptually, loading is simple. Loading a program or loading a handler to a certain namespace, in principle is the same. The loading of handlers into a namespace is carried out using *Add-Handler* to modify the *Workspace* of the namespace which contains not only the state of that namespace but the map and the handlers of that namespace. The loading of the PUT is more involved but shares many similarities with adding handlers to a namespace.

## Debugging Semantics

Semantic information that describes both structure and meaning to the namespace has to be supplied to the human user by the debug agent. Without appropriate "cues" to indicate meaning to the human user, results of requests made would be meaningless—in fact the original request itself would have no meaning to the human user. It is for reasons such as this that the Arch Slinky model of Human Computer Interaction has been adopted. While this user interfacing problem is not central to this research, some aspects are considered here. For hardware resources modeled by namespaces there is a need to map symbolic forms recognisable to the user of the debug agent (the debug client) requests to the namespace which models that resource, expressed perhaps in terms of a size and offset. At an even higher level, the task of mapping the user program to associated program resources such as variables and mapping of these to the objects that lie within a given namespace also needs to be considered.

> **Principle of Semantic Translation**
>
> *The debug agent should take care of the semantic complexities of the target resources with monitor design being simplified as a result. The target should therefore be modelled as an abstract machine.*

For each hardware card there could be a descriptor area (possibly stored in ROM) of each card that provided simple product information about the type of card such as who manufactured it. This information is used by the monitor and is communicated to the debug agent so that an appropriate hardware resource description (if it exists) can be loaded for that session to provide appropriate semantic interpretation and conversion between the abstract namespace form and the "concrete" symbolic form that describes the namespace on the debug agent. Many such symbol descriptors could exist for a debugging session with a monitor.

An implementation of such a system as described would work best when some form of auto-configuration of hardware existed. The most ideal hardware platform would allow dynamic configuration to occur as well as auto-configuration. The monitor could then relay all changes to target monitor's namespace to the debug agent. Corresponding symbol descriptors could then be loaded by the debug agent to provide some information regarding the resource added.

## Data Representation

Target configuration management is one of the major issues when considering distributed embedded systems. The namespace model is used to provide a method of solving the problems of configuration management of target systems, most notably the problems of dynamic configuration.

What also needs to be considered, however, are the problems of the heterogeneous domain. So far, discussion of namespaces has only considered the external properties, the symbols that compose the name interface that a namespace provides. The problems of *addressing* names and acquiring data from a namespace causes problems of a different nature that affect the specification of packet protocol and interface to handlers. These problems are due to the many ways that data is represented and addressed by processing units and reflect the difficulties of resource level debugging of heterogeneous systems.

If protocols are to become suitable for open systems use as discussed in Chapter 2, it is clear that such protocols must exist for handling the data representation in such a way that the two parties involved can communicate sensibly. The approaches taken to solve the issues of data representation vary from one protocol implementation to another. For example, the TCP/IP protocol uses a big endian byte ordering discipline for describing all sixteen and thirty-two bit fields in the protocol headers, little endian machines which implement a TCP/IP protocol stack must convert all header values to the correct byte order due to this.

The demands of the debug agent for resource level access are less demanding than that of having to support the generalised RPC functionality described in Chapter 2. Access to records or descriptors can be decomposed to a number of primitive accesses to a given namespace. Data types for the namespace protocol only need to be simple scalars. Resource access can be modeled as a series of data object accesses to a namespace. What is required is a means of representing requests to namespaces in a manner that preserves the meaning of the request across architectures. From the point of view of replies, reply data must be transmitted as efficiently as possible with the major aim of minimising data conversion. Assumptions of physical word size cannot be made by the protocol.

The data type, *machine word* is highly configuration dependent. Some machines may consider a word to be eight bits, others may consider it to be wider than this. The data types supported should be dependent on the processor configuration. The data representation protocol cannot limit itself to machines that model data in multiple eight-bit chunks. The era of machines which had ten bit or twelve bit bytes may be over but the data model should be able to support "unusual" non-octet oriented byte sizes. The main issues of machine data representation lie in three areas:—

a. The protocol request which provides a path to the correct namespace and handler must be defined in a way that is common to both debug agent and monitor server.

b. The handler interface has to describe the format of the parameters as they are sent from the debug agent (or possibly another monitor). The parameters themselves would be sent in a form understood by the namespace handlers.

c. Any responses from the handler must be *efficiently* sent back to the requester. The requirement here is to minimise or eliminate the need to convert data into a network byte order of any sort because the overheads for conversion may cause the monitor to affect the PUT's performance.

The solution used by the protocol is a hybrid of the ideas of Apollo NDR and ASN.1 described in Chapter 2. The solution is much like ASN.1 in the way that an *explicit typing* system is used to

describe types significant to the namespace protocol. The explicit type system "tags" data with a type prefix. Data associated with directing a namespace *Find-Name* operation such as the names that make up a namepath are described in an agreed upon form required when messaging between monitors and debug agent occurs. Parameters to a handler are tagged with an ARGUMENT tag, any following values are described in the native byte order of the machine so that handlers do not have to be responsible for time consuming data conversion. The solution is much like Apollo NDR in the way that data replies are handled. While the tags and respective data (except argument packs) use an agreed upon representation, replies are sent back to the requester in "raw" form. No compensation is made for byte ordering. There are two reasons for this:—

(1) Efficiency—the debug agent can always re-order bytes but it may be unfeasible for the monitor server to do so. Conversion takes time for each quantity transmitted in the reply.

(2) Data as returned from a namespace handler might be more useful to a human in *untranslated* form.

### Debugging at the Event Level

This chapter has so far only considered resource level debugging, the ability to inspect program and hardware *state* using the namespace model. This capability on its own is insufficient, the ability to monitor and control program execution is also important. Often, the task of debugging is that of measuring *progress* and the capturing of significant *events* in the software being debugged as they occur. Ultimately, a program decomposes down to machine level instructions that affect the state of the program. This change of state can be used to measure the amount of progress made toward a stable state being reached or termination. If a variable changes from one value to another, a change of state has occurred. The ideas of progress and events are linked. Events visible to the human user of the debug client are perceived in some way to show that the system is doing *something.* It may not be behaving correctly but the changes in state seem to imply that the program is executing.

An *event* is a change in state. The degree of state change depends on the "level of abstraction" at which one wants to examine the events. At the machine instruction level, an event might be the change in state that occurs when every instruction is executed by the microprocessor. To a person debugging the source code of a programming language such as Pascal, source language statements and changes of control flow are significant events. At an even higher level, events might be classified as the creation and destruction of execution objects such as processes and tasks, the sending and receiving of messages between tasks and the claiming and releasing of semaphores.

To debug at the event level requires a means of specifying events of interest in the system being debugged and a way of detecting the events and notifying the debug agent when an event occurs. Some systems such as the TMP Monitoring system [Hab90] use a hybrid hardware and software approach. To support such a system requires that each processor node has TMP hardware attached and may not be practical for a heterogeneous system.

The debug protocol supports event definition by the use of breakpoint capabilities which are readily available on microprocessors. Break events are special machine states which can be used to cause programs to halt their execution. Placing breakpoints in an application can be very useful due to the fact that a well placed breakpoint can measure the progress of a software system under test. When a breakpoint is reached the user can then examine the state to determine if the system is behaving as specified.

### Break Events

Debugging tools such as *gdb* and *Dbxtool* are commonly used to debug applications running on workstations. Both tools allow tracing and breakpoints to be set in application code. The type of breakpoint specifiable to the application under test is a *local breakpoint.* The term local breakpoint is used here to describe a breakpoint that only has effect within the process for which it is defined. Practically all non-distributed debugging systems support a local breakpoint. The effect of a local breakpoint is to halt the application in which the breakpoint is set when that breakpoint state is detected.

Distributed applications have a different architecture. If the application is truly distributed in the sense that application components require services to be accessed from remote machines, the local breakpoint becomes less useful. Debugging a non-distributed application is conceptually simpler because all the application state is available, this is not the case with distributed applications. However, the problem still remains that access to application state is required to debug the system. This state however is now distributed. A means of determining *consistent global state* is required. The local breakpoint is not capable of expressing this requirement because it assumes that all the application executes on one node. If global state is required then a means of taking snapshots of a consistent global state must be available to the debugger. The term "consistent" is used to qualify the fact that while local node states are easily and meaningfully frozen, obtaining an instantaneous global snapshot is impossible due to delays in message propagation through a communications medium. To provide mechanism to support the collection of global snapshots, breakpoints have to also be able to halt processes that are executing on other remote machines that have some relationship to the distributed processing task. Distributed debugging protocols must therefore support *global breakpoints* which allow related remote processes to be halted when break events at a particular processor node occur.

An additional concept is that of group breakpoints, which are useful for the purposes of halting only a certain group of tasks that are active while leaving others outside the group to continue unaffected.

### Activity Relative Break Events

While breakpoints have been successfully used to debug programs on single processor machines, the programs being debugged have represented a small, simple class of single-threaded applications for which the local breakpoint approach works. Distributed and multi-threaded applications represent a more complicated issue. The problem with breakpoints is that they are too low-level. For applications which use common code modules that are accessed by multiple tasks, the breakpoint does not provide sufficient granularity to identify the task which was executing when the breakpoint event occurred.

Breakpoints without some other support are an indiscriminant means of halting activity. Break events cause the application to halt when the processor goes into a breakpoint state. The other problem which results from the low-level nature of breakpoints is that often the user is interested in halting only when a particular task accesses that section of code, other tasks should be allowed to continue unaffected. A facility is needed that augments the usefulness of breakpoints as a general event notification mechanism with the ability to discriminate between tasks that are affected by that breakpoint and those that are not. Providing a facility that allows task level debugging to be effected is essential to the development of distributed applications.

Support for concurrency and distributed debugging is provided by an *entities* structure. The term entity was used earlier in this chapter to describe namespaces which did not directly describe

hardware resources. Particular entities mentioned previously were called *monitor entities* because they described the various monitor interfaces that allowed program execution to be controlled, sessions to be managed and breakpoints to be set—what might be described as "typical" monitor or debugger functionality. *Program entities* describe logical execution structures at a higher level than that of just a machine instruction stream. These structures include those that are a part of the run-time environment such as processes, co-routines and other active data objects. The entity concept requires some information to provide a means of determining *which* task was executing when an event occurs. The advantage of an entity is that it provides information that allows greater discrimination when debugging a system of multiple tasks. A code section that is shared between multiple tasks can have a breakpoint set in such a way that only one task or a particular group of tasks are halted, not any task which would be the case with a breakpoint.

As described, the entity and breakpoints mechanisms must be related to each other. Breakpoints describe points of interest where program execution *may* be halted, entities allow a decision to be made as to which task or task group running in a program *is* halted. Breakpoints based upon entities are considerably more powerful due to the fact that only the entities which are suspected of being faulty have to be halted. The ability to isolate an erroneously behaving task in the program is important. The expressive power of entities is further improved if entities can allow the halting of tasks on remote machines which are executing tasks that are related in some way to the local machine. For example, this may be of great use when debugging a communicating tasks group.

The entities model uses the namespace model to support breakpoint functionality. A program entity is just another monitor namespace which can be created and destroyed during the lifetime of the PUT. The difference is how entities get created and destroyed and how the break handler uses entity namespace to discriminate between tasks that should be continued and those that should be halted. Entities represent logical objects of execution, in a Modula-2 program these objects might be co-routines or processes, in Ada they could be tasks. When a program is downloaded to the target hardware for testing, entities *may* be *registered* each time an object such as a co-routine or process is created. While entities represent *active* objects such as tasks, the entities themselves are passive much in the way that resource namespaces are passive. Registration often occurs at initialisation but can occur anywhere and at anytime throughout the life of the program under test. The registration of entities can only occur if there is code present to register the entities. For example, when a process is created in a Modula-2 program, the entity describing that process could be created. When the process is destroyed, the corresponding entity (provided it exists) is also destroyed.

NOTE Entities are *NOT* processes or tasks! They only *describe* tasks. The monitor server is incapable of using this information to schedule task execution. That scheduling capability is part of the PUT and is not the responsibility of the monitor.

The entity and device share considerable common ground; both share the same requirements for dynamic configuration. For a device, dynamic configuration is the creation or destruction of hardware objects. Entities are dynamically created by the program as it executes and as the program executes to completion, entities are destroyed.

The structures to support entities are based on the active data objects that the entities represent. Each entity has a local workspace, an execution context and some means of identifying the entity. The exact mechanism by which entities are determined to be active in addition to their relationship with breakpoints is discussed in Chapter 5.

While the monitor is capable of registering and deregistering entities that describe the task structure of a user program, the monitor does not know what the entities describe. Only the

debug client and the human user have knowledge that the entity created actually represents a "task" running within a program. The monitor can determine that an entity is associated with a breakpoint, *what* that entity represents (co-routine or process) is only known by the human debugger or possibly the debug client.

## Summary

The monitor is a distributed application much like the programs-under-test which it is responsible for hosting to allow the process of debugging to take place. As a distributed client-server application it is controlled by a communications *protocol.*

Because the monitor exists at the lowest layer as some primitive operating system kernel there are no convenience services available for use by the monitor. This means that the monitor must provide such facilities for itself. The monitor however, exists so that applications may be developed in such a manner that they can assume the control of the machine. Ultimately this means that the design of the monitor is constrained in such a way that the protocol is kept as simple as possible and that resource usage is minimised.

Given the requirements of a distributed debugger described in the previous chapter the *namespace model* was introduced. It is used to implement monitor services such as controlling program execution as well as implementing access to hardware resources. The namespace model is an interconnected tree of monitor and hardware resources. Access to a particular resource is specified through a namespace path to a named handler and arguments.

# Chapter 4

# Namespace Model

This chapter develops the model which is central to this thesis—the namespace. The namespace model is developed from a structural and behavioural perspective using the *Vienna Definition Method Specification Language (VDM-SL)* to formally define its semantics, full details of which are given in Annex A.

### What is a Namespace?

A concept which is central to the design of the monitor is that of a *namespace environment.* A namespace environment can be viewed as an association between names within a particular environment mapping to:—

a. an action to be performed within that environment or

b. another subordinate namespace environment.

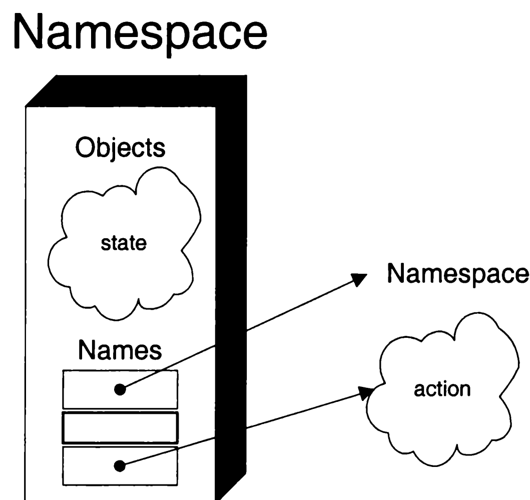



Fig. 4.1 *Namespace—Objects, Names and Actions*

The abstraction models the construction of debug agents and monitor servers which are capable of being *dynamically* configured. The model therefore contrasts with traditional debugging systems that have a static view of configuration. Dynamic configuration functionality requires a facility for construction and removal of environments and their associated names at run-time.

> **Definition—Namespace**
>
> *A namespace consists of a workspace together with a map of names to actions that manipulate that namespace or other namespaces.*

NOTE The following simplified definitions are given in the body of the text to demonstrate the derivation of the namespace model from first principles. The actual definitions used in the final form are given in the annexes.

types

*Message-Kind* =
⟨*NAME*⟩ |
...; – further tags described in Annex A.

*Name* =
token;

By itself, a *NAME* tag as defined by the *Message-Kind* type is insufficient to denote more than one unique mapping. The *Name* type is used to denote a unique name in the domain of that namespace.

The namespace definition now becomes a mapping of a token (whose type is tagged *NAME*) to a handler action. Multiple *unique* names map to handler actions that *may* be unique.

types

*Namespace* =
$Name \xrightarrow{m} Handler$;

*Message-Body* =
$token^{+}$; – further message body types described in Annex A

*Msg-Element* ::
*kind* : *Message-Kind*
*value* : [*Message-Body*]; – invariant for this type described in Annex A

## Handler Arguments

Arguments to a handler are supplied as a *Msg-Element*. The arguments supplied in the *value* component are known by the debug agent and invoked handler.

types

*Argument* =
*Msg-Element*
inv *mk-Msg-Element(kind,_)* $\triangleq$
*kind* = ⟨*ARGUMENT*⟩;

It is the responsibility of the *Finder* default handler to take a *Name* and *Argument* type and invoke the handler action corresponding to the name with the correct associated argument pack.

## Handler Execution

When the handler completes, a result indicative of the success of the operation is returned.

types

*Handler-Status* =
⟨*SUCCESS*⟩ |
⟨*FAILURE*⟩;

a. *SUCCESS* describes the result of an operation that yielded a successful result (as determined by that operation).

b. *FAILURE* describes the result of an operation that yielded an unsuccessful result (as determined by that operation).

## Handlers

Namespace handlers have been defined as acting on the workspace area contained by the namespace. Such a handler is defined as being a *Work-Handler.* A *Work-Handler* takes an *Argument* and *Workspace* and returns some result and a *Handler-Status.*

types
*Workspace* ::
*base* : *Address*
*size* : $\mathbf{N}_1$
*ws-state* : *Env-State;* State area of namespace environment
*Work-Handler* =
*fun* : *Argument* × *Workspace* → *Data* × *Handler-Status;*

## Environment

The revised definition of the *Namespace* now becomes:—

*Env* ::
*workenv* : *Workspace*
*map* : *Name* $\xrightarrow{m}$ *Work-Handler;*

The final model must allow for a multiple number of namespaces, the model skeleton developed thus far can only define multiple unconnected namespaces. The namespace model requires the interconnection of namespaces to be possible. Modelling concepts such as dynamic configuration then becomes possible—namespaces can be constructed and connected together or removed and disconnected as required. To do this requires that the definition of the handler type *Work-Handler* is expanded.

While a *Work-Handler* performs namespace related services such as establishing a session, setting a breakpoint etc, the range of the mapping should be extended to mapping to another *Namespace.* By extending the skeleton model to map names to other namespaces, interconnections of namespaces become possible.

The debug agent and monitor can invoke an action (henceforth referred to as a handler) to operate on that namespace by supplying the name of the handler to be invoked along with handler arguments required by that handler. As namespaces are arranged as a single rooted $n$-ary tree, access to namespaces within the namespace tree may require that messages be passed between namespaces.

A possible definition of this could be:—

types
*Name-Result* =
*Env* | *Work-Handler;*
*Env* ::
*workenv* : *Workspace*
*map* : *Name* $\xrightarrow{m}$ *Name-Result;*

The purpose of a *Work-Handler* is to manipulate the workspace of the namespace of which the *Work-Handler* is a member. For a namespace to be able to create and destroy other child environments (*Env*) below it another type of handler must exist that is capable of manipulating the entire *Env.* This handler type is known as an *Env-Handler.*

An *Env-Handler* takes a *Message* and an *Env* yielding a *Message* (similar to *Handler-Status*) and an *Env.*

types

*Name-Result* =
*Env* | *Work-Handler* | *Env-Handler;*

*Env* ::
*workenv* : *Workspace*
*map* : *Name* $\xrightarrow{m}$ *Name-Result;*

The *Env-Handler* is essential to the namespace model because it provides the framework necessary for the *interconnection* of namespace environments. It opens the model by making the model *extensible.* This is because the *map* is modifiable by the *Env-Handler.*

### The Default Environment

What handlers should be defined as being *Env-Handlers*? Any handler that has to manipulate the environment, or more specifically the *map* within an environment, should be an *Env-Handler.*

For each namespace, the following default *Env-Handlers* should exist:—

a. A directory find and invoke service—a *Find-Name* handler responsible for resolving names and invoking *Work-Handlers* if necessary.

b. A namespace environment addition service *Add-Name* to establish child namespace environments subordinate to the current namespace.

c. A namespace environment removal service *Remove-Name* to remove child namespace environments subordinate to the current namespace.

d. A *Work-Handler* addition service *Add-Handler* to establish new handler services within the current namespace.

e. A *Work-Handler* removal service *Remove-Handler* to destroy a handler within the current namespace.

f. A *Work-Handler* replacement service *Replace-Handler* that changes the current handler's behaviour but not the name of the handler itself.

All environments have these *default environment* handlers as the basis of the namespace. However, as discussed in Chapter 3, the *Find-Name* handler is the most important to the namespace model. Because it acts as a name resolver and dispatcher, the *Find-Name* handler interconnects namespaces. All other handlers, irrespective of whether their type was *Env-Handler* or *Work-Handler*, would be useless without the resolution and dispatching abilities of the *Find-Name* handler.

From the above discussion the default environment map for a namespace may now be given as:—

values

$\rho_{default}$ : *Map* = {
$\langle ADD\text{-}NAME \rangle \mapsto$ *Add-Name*,
$\langle REMOVE\text{-}NAME \rangle \mapsto$ *Remove-Name*,
$\langle ADD\text{-}HANDLER \rangle \mapsto$ *Add-Handler*,
$\langle REMOVE\text{-}HANDLER \rangle \mapsto$ *Remove-Handler*,
$\langle REPLACE\text{-}HANDLER \rangle \mapsto$ *Replace-Handler*
};

The *Find-Name* operation is mapped separately to the default map and as such does not appear in the above definition. The final form of the *Env* structure can now be given:—

types
  *Env* ::
    *ws* : *Workspace*
    *mapping* : *Map*
    *my-name* : *Name*
    *finder* : *Env-Handler;* This is the Find-Name handler

## Namespace Model Closure

A model is said to be *closed* if the application of operations yields results that lie within that model. Is the namespace model closed?

Operations on a namespace environment can only be performed by having a reference to the environment that performs that operation. All namespace handlers are bound to a namespace environment. The *Find-Name* handler is the only legitimate means of accessing these handlers. By having a mapping between name and handler it is more difficult to perform services without the aid of the *Find-Name* handler.

Because *Work-Handlers* operate only on the *Workspace* associated with the namespace, *Work-Handlers* can only ever have local effect. Because *Work-Handlers* do not have access to the map of names, their effect is restricted to the state established within that namespace.

The addition, removal or replacement of handlers for a given environment only affects that namespace and no other. However, namespace construction and destruction do have an effect on the current environment and child environments. This namespace construction and destruction still results in the same model, albeit larger or smaller depending upon whether a namespace was added or removed.

The namespace model is therefore closed with respect to:—

(1) Handler access via *Find-Name*.

(2) The restriction of *Work-Handlers* to having only local effect within the namespace that the *Work-Handlers* are a part of.

(3) The addition or removal of namespaces. The model just becomes "larger" or "smaller" as required.

## Handlers—In Practice

A namespace handler in practice should behave like a procedure call. The handler has a name which may be thought of as an identifier for the operation that the handler performs. Refer to Figure 4.2.

The passing of arguments to a handler may be achieved in a manner that to the handler itself is no different from a procedure call. This means that arguments should desirably be packaged in a format that allows them to pushed onto a stack so that the handler can access the parameters and perform the appropriate operations.

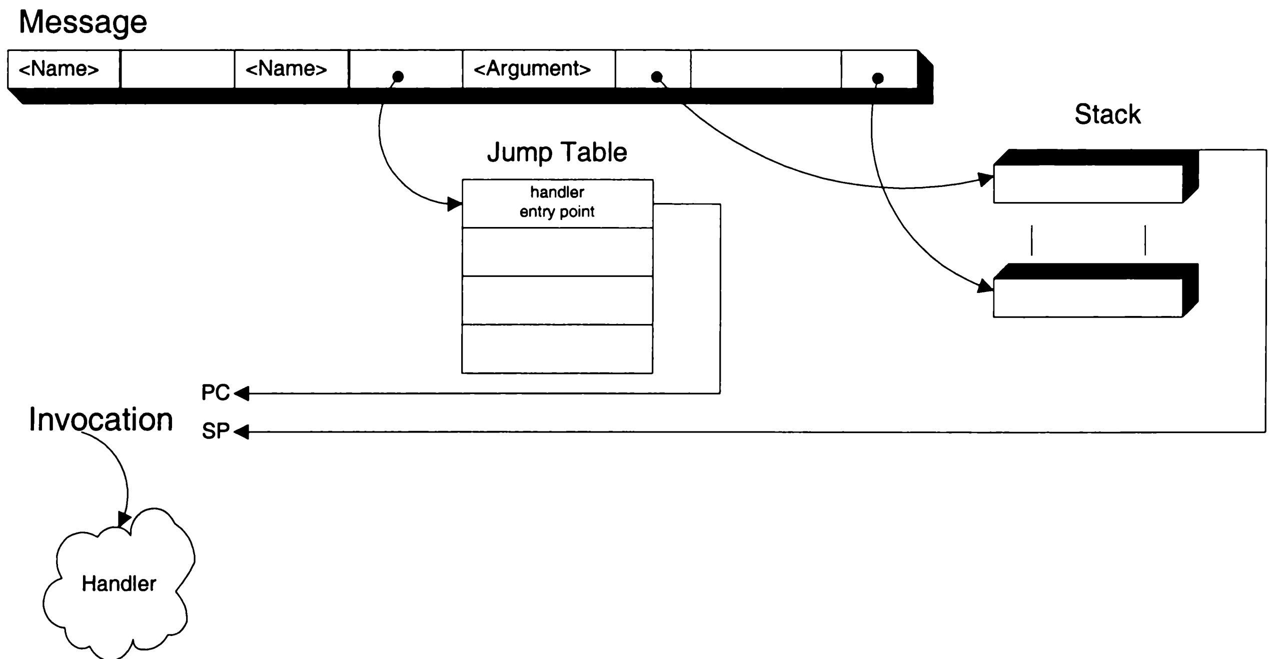



Fig. 4.2 *Handler Messages vs. Handler Invocation*

The general approach of how handlers could be made to work can be viewed in a similar way to approaches taken by remote procedure call (RPC) architectures such as SUN RPC and Xerox Courier. The only choice that will govern the approach taken in an implementation will be the overall requirements for the efficient marshalling and unmarshalling of handler arguments. The less computational effort required by the monitor to dispatch to a handler the better likelihood of realising the desired goal of minimising invasiveness.

The issue of which form the arguments take actually depends on whether the handler is a *model handler* or a *user-defined* handler. Model handlers such as those described in this chapter and the next require type concretions that have consistent meaning between target configurations especially since the source of these messages could be a debug agent, or in the case of an external breakpoint message, another monitor! Such types include those that define the breakpoint identity and break addresses used by the break handler. These types have a defined byte ordering that is discussed in Chapter 9. User defined handlers can take a variety of argument types, the arguments and their results are given in native form. This implies that *minimal* data conversion *may* need to be done for some *model types*, while user defined handlers and arguments may take an arbitrary form whose resultant replies are in an untranslated form when received by the debug agent.

### Default Environment Handler Specifications

The implicitly specified default environment handlers identified earlier (and formally defined in Annex A) are a part of *all* namespace environments:—

(1) Find-Name

(2) Add-Name

(3) Remove-Name

(4) Add-Handler

(5) Remove-Handler

(6) Replace-Handler

*Env-Handlers* are constructors of the namespace environments as well as destructors of namespace environments and their respective handlers. These are sufficient to construct and destroy environments as well as augment the behaviour of the space by means of manipulating handlers within the current environment. In addition to these default handlers, there are handlers that "characterise" a particular namespace. For example, a breakpoint namespace includes the default handlers as well as service handlers which provide support for manipulating breakpoints.

An *Env-Handler* can act on both the namespace map and the workspace of an environment. Each environment is completely separate from every other. No workspaces are shared. Communication between *Work-Handlers* takes place via messaging between namespaces. This is of considerable benefit to the overall security of the system—communication can only take place through the visible handlers of that environment, no "back-door" techniques are possible.

The following sections describe the six *Env-Handlers* listed above that are always a part of the namespace environment instance.

## Finding Names

The namespace model operates by passing messages from some external entity (normally the debug agent) to the namespace "engine" which is a part of the monitor. The mechanisms used to pass messages into the namespace model are described in Chapter 6, but assume for the moment that a message has become available for processing by the model:—

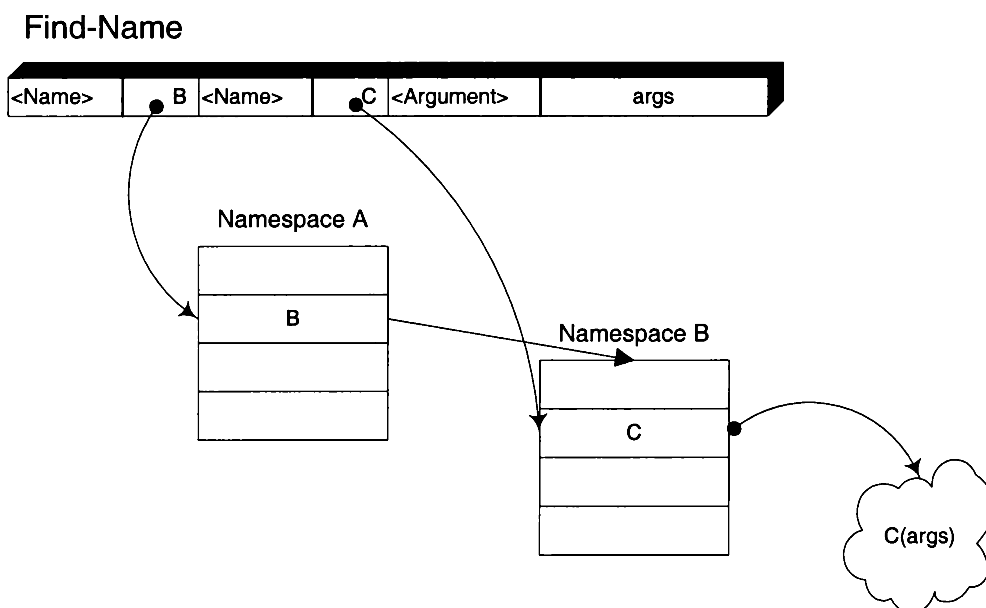



Fig. 4.3 *Finding a named handler within the namespace framework*

Because the namespaces may be nested within another namespace to arbitrary depth a multiple number of message elements (*Msg-Element*) may be required to specify the namespace handler to be invoked. A *Message* may now, therefore be defined to be a sequence of one or more message elements.

types
  *Message* =
    *Msg-Element*$^*$;

Messages consist of a finite sequence of *Msg-Elements.* Because namespaces can be nested to an arbitrary depth, the message consists of a chain of *NAME Msg-Elements* followed by an *ARGUMENT Msg-Element.*

The structure of messages and the interconnection of namespaces themselves implies a recursive processing of names by each namespace. The basic task of the *Find-Name* operation is to "find" a name which matches against the current namespace *map* applying the following rules:—

(1) The *Find-Name* operation always operates on the head of the message (the *Message* head element).

(2) If the first element is a name and it maps to another environment (a child namespace) then the tail of the message is passed to the child namespace so that the *Find-Name* of the child namespace can be applied to the rest of the message (the *Message* tail).

(3) If the first element is a name and it maps to an *Env-Handler* then the tail of the message is passed to the local environment handler.

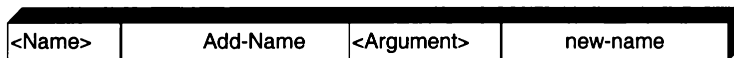
(4) If the first element is a name and it maps to a *Work-Handler* then the remainder of the message (the arguments) should be applied to that local work handler.

(5) If in any of the above cases the first element is not a name or its value is not in the domain of the map then return an error.

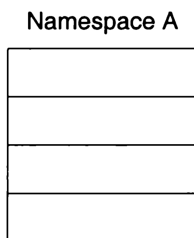
## Adding Names

Adding a namespace (using *Add-Name*) and attaching it to the current environment is the one operation that enables the namespace model to be extended as requirements dictate. Next to *Find-Name*, the *Add-Name* handler may be considered as the most important handler since it is a *model generator* allowing namespace interconnection to take place.

Add-Name




Before




After

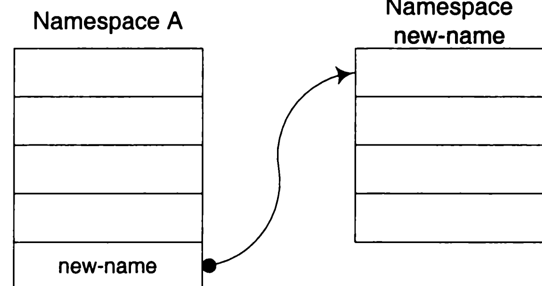



Fig. 4.4 *Adding a namespace to an environment*

The steps required to add a namespace are as follows:—

(1) Determine whether the name to be added already exists as either an *Env*, *Env-Handler* or *Work-Handler*. If the name cannot be found then the following two steps can take place.

(2) Create a new environment and populate it with the default components.

(3) Add the name to namespace environment maplet to the map of the namespace that constructed the child namespace environment.

## Removing Names

Removing a namespace (using *Remove-Name*) is just the inverse of *Add-Name*. The removal of a namespace results in all subordinate namespaces that are in the the same path being removed.

The steps required to remove a namespace may be summarised as:—

(1) Check if the namespace environment to be removed actually exists and is a namespace environment. If the name does not exist then return an error result.

(2) Remove the child namespace environment.

(3) Modify the map to reflect the namespace maplet is no longer in existence.

Because a namespace may have an arbitrary number of subordinate namespaces, a concrete implementation of this handler will need to consider removing more than just the name and mapped environment. In practice, the removal of an environment will require the recursive removal of all subordinate environments.

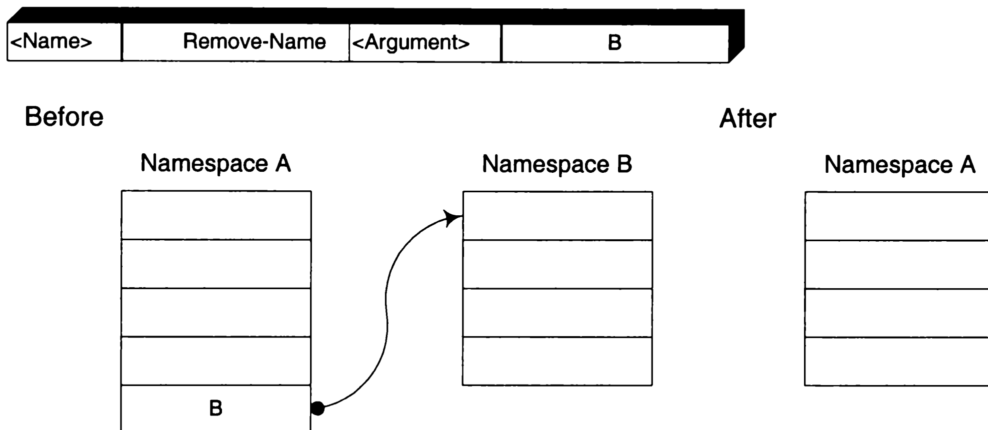



Fig. 4.5 *Removing a namespace from an environment*

## Adding Handlers

The ability to add a *Work-Handler* (with *Add-Handler*) is as important to the namespace model as the ability to locate handlers and add new namespaces. The ability to add a handler enables the extension of services provided by the namespace.

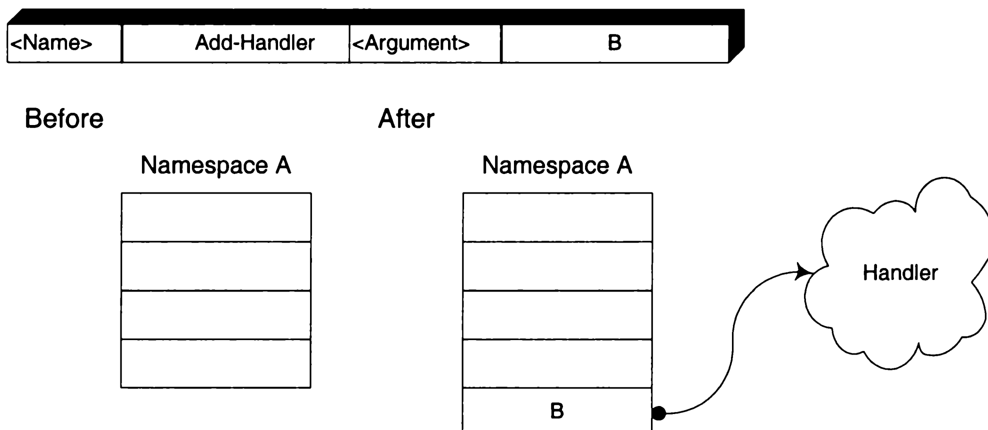



Fig. 4.6 *Adding a handler to an environment*

The steps required to add a new handler to a namespace may be defined as:—

(1) Determine whether the name already exists. If the name does exist regardless of whether it is an *Env*, *Env-Handler* or *Work-Handler* then the addition of the handler shall fail and an error shall be returned.

(2) Instantiate the handler.

(3) Modify the map of the environment in which the *Work-Handler* is being added.

## Removing Handler

Much like the *Add-Name/Remove-Name* inverses, the removing of a *Work-Handler* previously added is accomplished by *Remove-Handler.*

The process of removing a *Work-Handler* is analogous to that required for removing a namespace except that *Remove-Handler* works only within an environment.

Remove-Handler

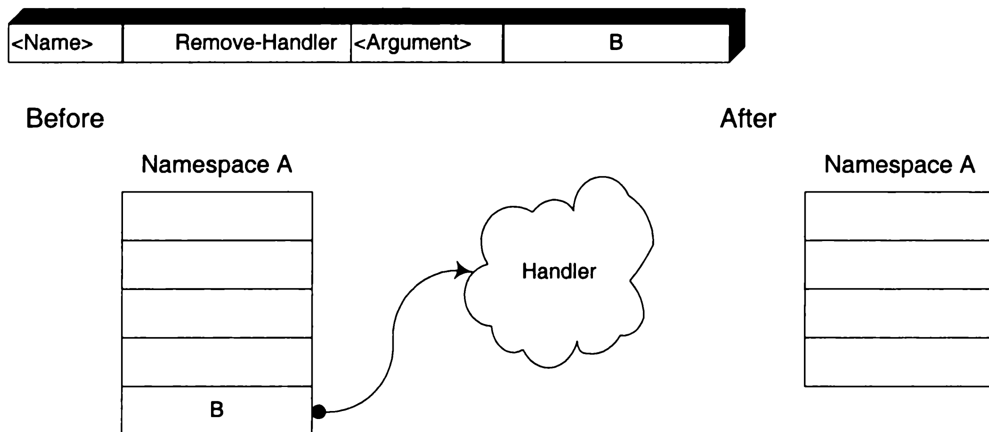



Fig. 4.7 *Removing a handler from an environment*

The steps required to remove a handler are as follows:—

(1) Determine whether the name maps to *Work-Handler* only. If it maps to a different type of handler then return an error result.

(2) Remove the handler.

(3) Remove the maplet of name to handler that previously existed.

## Replacing Handlers

In some instances, the name of *Work-Handler* may need to be retained but its corresponding handler be replaced.

Replacing a handler may be viewed as a combination of the addition and removal of a handler.

(1) Determine whether the name to be replaced actually exists in this map and whether it is a *Work-Handler.* If it does not then return an error result.

(2) Instantiate the new handler and remove the previous handler mapped to by the name.

(3) Override the previous mapping of that name with the new handler mapped to that name.

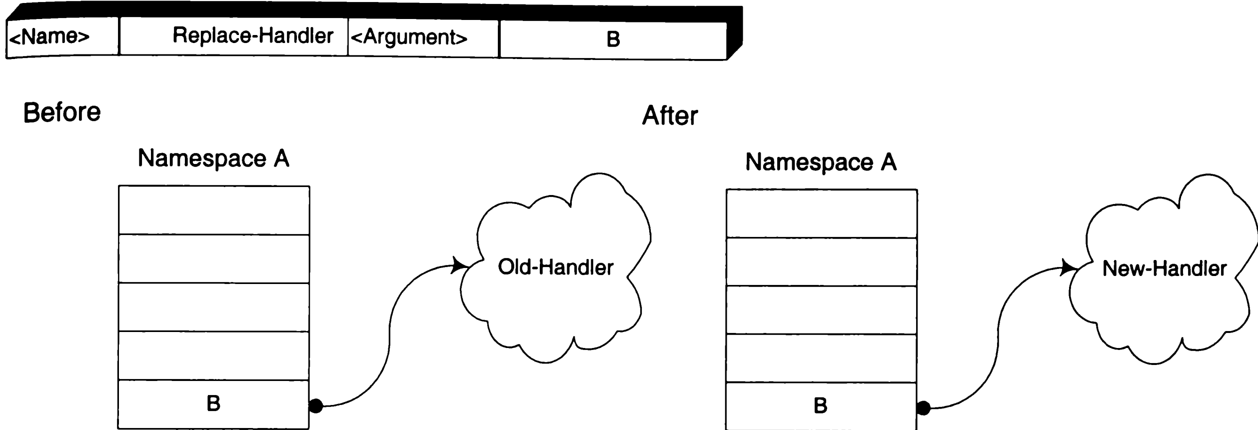



Fig. 4.8 *Replacing a handler within an environment*

## Dynamic Modification—In Practice

The namespace model is dynamic. The monitor can shrink or grow during the lifetime of a session between debug agent and monitor. The ability to add and remove both namespaces and handlers allows the monitor software to be considerably more flexible. This is best demonstrated by events such as dynamic configuration of hardware resources where addition or removal of devices can cause corresponding namespaces to be added or removed.

The namespace model is useful for more than just modelling hardware resources. *Every* monitor service is a handler invoked within a given namespace. Namespaces can also model abstractions such as breakpoints and session connections. Being able to modify handlers such as the breakpoint handler, means that the monitor's behaviour can be altered when handling events. However, this is subject to the constraints of the calling interface between message sender and receiver. Assuming this requirement is met, the monitor and the testing of the PUT can be achieved by custom configurations that may not be provided by a debugger with fixed methods of debugging. New debugging techniques can also be tested and compared using this ability to modify namespace handlers on-the-fly. Special debugging methods can be used for certain situations without having to "hard-wire" these methods to the monitor code.

How could dynamic alteration be realised in an implementation? To manage handlers being added, removed or replaced requires that the monitor has to manage storage space. A monitor will need to allocate and deallocate memory for namespaces and handlers as these respective objects are added or removed. There are a number of means by which this could be achieved but one method that could be used to manage the dynamic addition of namespaces and handlers would be to have the monitor allocate storage space for a namespace on creation, any handlers added or removed could be managed from within this space. This has the advantage of being able to remove namespaces easily—handlers are an inclusive part of the namespace. The mapping of names to handlers can then be refined to a name mapping to a code entry point. This has implications for implementation of handler code—it must be position independent as the handler code could be placed anywhere in the address space managed by the monitor.

## Limitations of Handlers

The namespace model's biggest advantage is its flexibility. Anything of benefit to the process of debugging a program can be added as a handler within a namespace. In practice the actual types

and numbers of handlers is limited by "reasonable" debugging features. Nonetheless, this is due *solely* to the nature of the tasks being performed when debugging is taking place.

The flexibility thus offered requires that one major rule must be followed:—

> *External requests or internal events cannot cause handler cycling or mutual recursion to take place by means of a handler sending messages to itself directly or indirectly.*

Handlers must eventually cause external message replies to be generated which may be sent to either debugger or potentially to other monitor servers. The message chain *must* cause an eventual response to the debug agent.

Another issue of concern is the situation where a handler is interrupted *during* its execution. The major problem in this situation is that of *fake updates* occurring. This kind of corruption can only take place when a handler within a namespace that alters state gets interrupted and that same handler (or possibly one that manipulates the same state) gets executed after the interrupt occurs.

Such a situation will not normally cause problems. When the monitor is active, it processes requests that have been *enqueued* by message sources such as traps and interrupts. Other message types are processed synchronously as they arrive, any NAME requests are *enqueued* to the monitor. If a handler that was dispatched by the monitor is interrupted, any state that the handler relies upon can be assured of remaining uncorrupted because the interrupter cannot invoke handlers that are not part of the namespace used for managing the communications link between debug agent and monitor directly—it can only do that by enqueuing a message to the monitor. Messages are processed in FIFO order by the monitor, the newly enqueued message (from the interrupt handler) "waits its turn" on the queue. When the interrupting co-routine transfers back to the co-routine that was interrupted, in this case the handler running as part of the monitor, the handler runs to completion, replies and the monitor acquires the next message on the queue—possibly the message that was enqueued by the interrupt handler that interrupted the handler execution previously.

The only situation where a handler interruption is not allowable is when *Send* is executing. As output messaging is *not* enqueued in the manner that the monitor input message queue is, potential corruption could occur if the monitor *Send* handler was interrupted during a message send operation, therefore:—

> *The operation that Send performs must be an atomic action.*

## The Nature of Workspace

All namespaces have state that provides *persistent* storage. This state area, known as a *Workspace* represents an area allocated during creation of the namespace used for maintaining state information possibly required by the handlers of a particular namespace. In addition to this is the namespace map and an area within the Workspace where the handler code may be loaded.

How the workspace is structured is up to the individual namespace concerned. Interpretation of the contents of workspace depends entirely on its function with respect to the object which it models. For example, the Target namespace structures part of its workspace as a breakpoint table with various fields within this table being suitable for the handling of breakpoints.

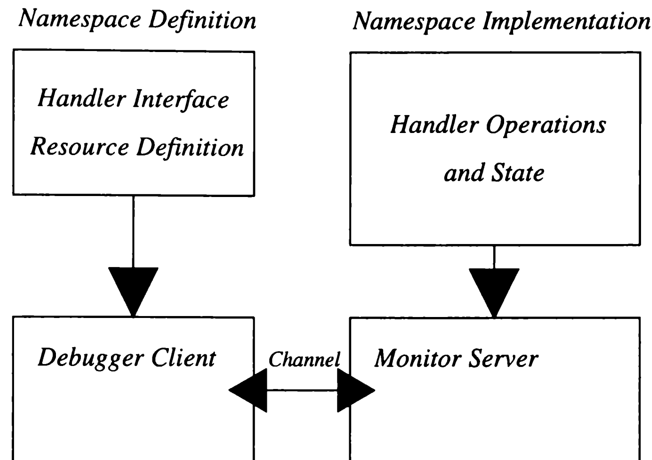



Fig. 4.9 *Namespace handlers are implementations, Namespace symbol descriptions are definitions*

It is useful to consider that a namespace and its handlers are much like a class body. The class interface definition resides on the debug client/ agent as a symbolic description of the hardware resource or monitor software objects. Workspace is created when a namespace environment is created. The workspace is destroyed when the namespace environment is removed. Between the time at which addition and removal of the environment takes place, handlers may be added to that namespace as well as any child environments. Handlers within a namespace act as "brokers" of that resource. The handlers provide an abstracted access mechanism to the resource, how the handler accesses or manipulates the resource is immaterial. The calling interface of the handler only supplies the necessary details.

## Implementation Considerations

The discussion of the namespace model has described only one means of invoking a handler within a particular namespace. This method involves the enqueuing of a message which gets successively dispatched by the *Find-Name* function within a namespace environment until the appropriate name has been found and further resolved/invoked or no name has been found. The messaging is effected by the use of the global message queue. Because of this, the handler invocation mechanism is *asynchronous* since the "coupling" mechanism between invoker and invokee follows a FIFO ordering. This ordering is acceptable for a large number of situations where the monitor is responding to messages issued by the debug agent or possibly other monitors. For example, a query to a memory namespace by the debug agent results in a message being enqueued upon reception by the monitor to only be acted upon when that request is at the head of the queue. Any intervening messages have to be processed before the above request.

The monitor can use message passing via the global message queue instead of the traditional procedure call mechanism on the occasion where it is necessary to do so—the breakpoint handler is an example of this. Communication between handlers of various namespaces therefore is accomplished indirectly by placing an appropriate message on the queue. Replies to requests do not get sent directly to the caller as the results are not passed back by reference or by typical parameter passing methods. Instead, replies get sent to a handler within the requesting namespace which acts as a *responder* for that type of message. In the practical sense, this means that tasks that involve two or more namespaces are accomplished by means of handlers acting in a pipelined fashion with the global message queue acting as the pipe with the "namespace engine" and its nested dispatchers processing requests. Any replies that need further processing by another namespace's handler (or

even a handler in the same namespace) do so by enqueuing those replies as arguments to the next handler.

While pipeline processing is possible in this model, most of the handlers described in this chapter and the next do not use this mechanism. Instead, a more direct means is used, the *synchronous* handler call. This is particularly the case where "instantaneous" non-blocking action is required and the asynchronism introduced by the queue is a liability for the action to be performed.

For handlers that exist as part of the monitor namespace environment the handler interface can be treated as a set of procedure entry points to be used *only* by the monitor. After all, a map either "points to" a procedure (handler) or to another name environment. This equivalent view of handlers allows a synchronous handler calling mechanism between parent and child namespaces that can be used by the monitor internally. The main reason for the synchronous calling mechanism is that some monitor services which require immediate response such as sending replies to the debug agent cannot be done asynchronously. The process of replying to a message from the debug agent must take place as soon as the reply is constructed. In effect, responses should be able to pre-empt the queue mechanism in the interests of efficiency. To obtain this functionality, one major rule has to be followed:—

> *The synchronous call mechanism must be provided with the environment of the handler before an invocation of that Work-Handler or possibly Env-Handler can take place.*

The requirement of the relevant namespace environment being provided is used to invoke the *Find-Name* handler of that environment. It is up to the finder of the environment to dispatch to the handler and to return results to the caller. In this way, the handler (Env-Handler or Work-Handler) behaves as a *local message requester and receiver* much like the debug agent and monitor behave as client and server. For example, a *Work-Handler* needs to invoke a service from another (subordinate) namespace.

The following diagram shows the similarities between a handler being called asynchronously via the queue and synchronously via a direct call to the handler entry point.

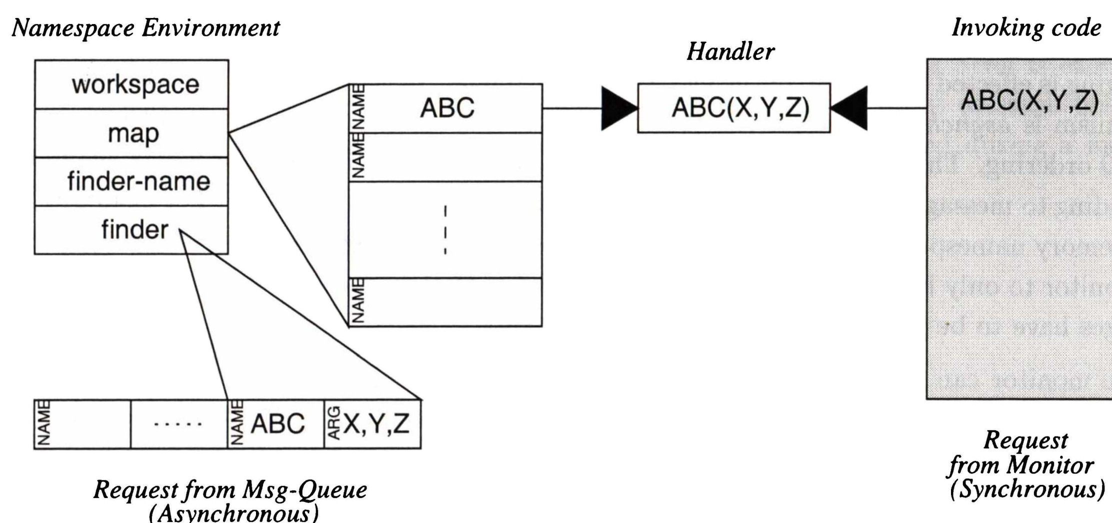



Fig. 4.10 *The namespace model allows synchronous and asynchronous handler invocation*

While the monitor can use the synchronous handler call mechanism, the debug agent specifies requests as a namespace path to the resource handler required—this means that such calls are inherently asynchronous because as they arrive at the target they are enqueued for processing by

the monitor. There may possibly be intervening requests that have to be responded to before that particular request is responded to due to FIFO queue processing.

A synchronous namespace handler invocation is constrained by the environment to which that handler is associated. The effect of calling a handler synchronously is exactly the same as calling it asynchronously—the handler can *only* act upon the state area of the namespace environment to which that handler is bound.

## Summary

This chapter developed the core of the namespace model using *VDM-SL* to specify structure and operations of the fundamental model. The base model defined in this chapter is used to model all monitor services providing a uniform means of service invocation as well as allowing for dynamic configuration of service.

# Chapter 5

# Specific Namespaces

This chapter describes *applications* of the namespace model for the purpose of providing debugging functionality. This debugging functionality is provided by services of the monitor to be accessed by the debug agent via some debug client/utility. The following diagram illustrates the namespace services provided by the monitor.

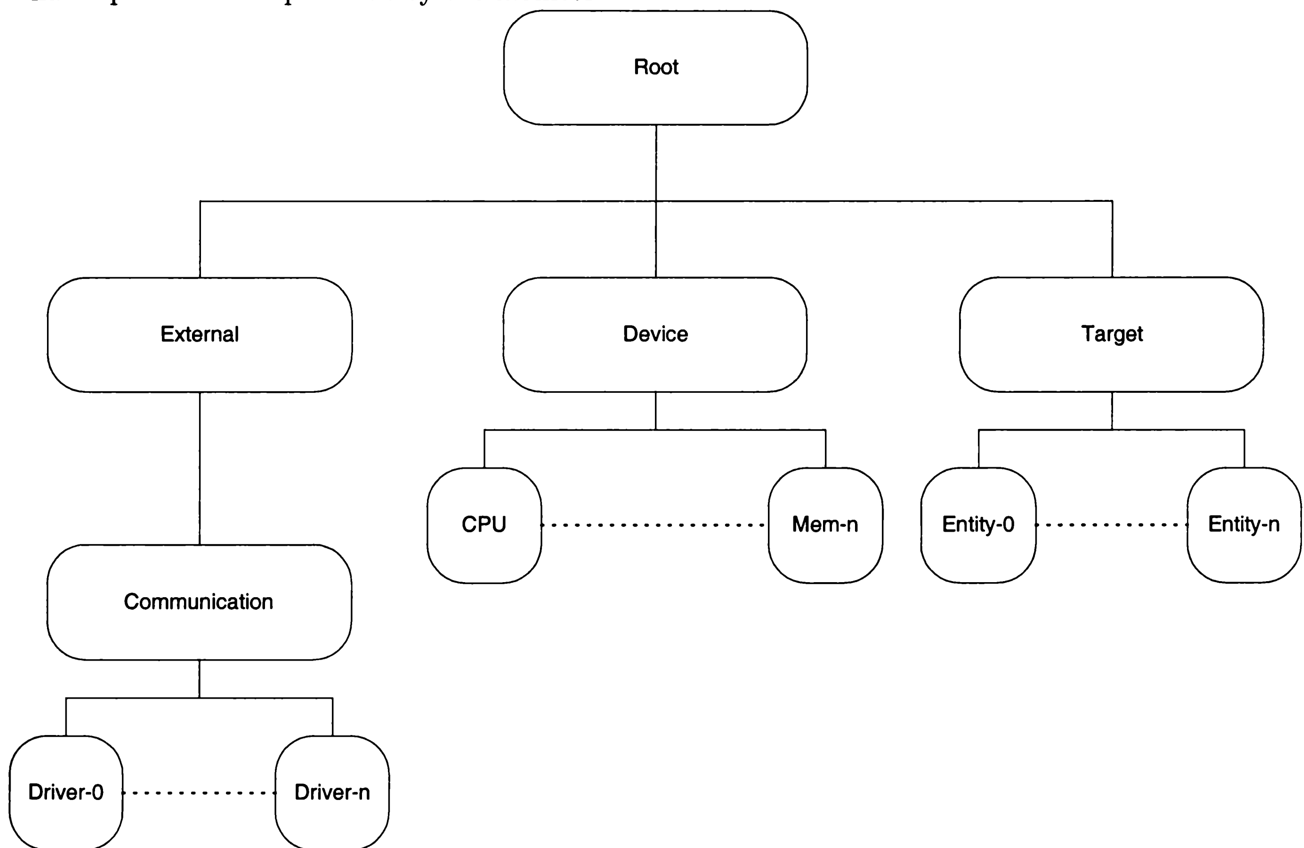



Fig. 5.1 *The monitor namespace framework for services*

The abstracted services described in this chapter form an interconnected namespace framework. This framework is static in the sense that it provides the basis of monitor functionality. However, the debug agent or the monitor itself (for example, in response to events such as dynamic configuration) can extend this framework.

The abilities to handle dynamic configuration and user customisable namespaces and handlers therefore come at no extra cost. This extensibility also allows for potential retargetting of the debug agent to a monitor. It is up to the debug agent to map higher level structured descriptions to actual namespace requests made by the debug agent. The same applies to replies made by the monitor when events occur. Since the debugger/debug agent has no fixed view of the target configuration it is possible to design a debugger that is *dynamically* retargettable. It could also be said that the namespace model and the nature of dynamic configuration makes this fixed view of

debug agent design untenable, the monitor and debug agent software *have* to become dynamically retargettable.

The benefit of dynamically retargettable debuggers is that only *one* debugger executable is required to debug a *group* of *heterogeneous* machines connected together. This is in contrast to the current situation where a multiple number of debuggers is required for each target machine—static retargetting is required for each configuration. This may work for homogeneous development environments but does not scale well for distributed heterogeneous environments.

### Root Namespace

The top of the namespace tree known as root namespace environment ($\rho_{root}$) is a host environment that contains all other namespace environments. The root namespace contains no *Work-Handlers*, but does contain all subordinate environments.

Root namespace is important because it is the *interface* point between events and messages such as traps, interrupts or debug agent requests and the monitor's debug functionality which is implemented using the namespace model. Anything above root namespace lies in the realm of the *functional core adapter* shown in the Arch-Slinky diagram (in Chapter 1). Anything below root namespace lies within the *functional core* of the monitor which uses the namespace model.

The remaining sections of this chapter describe the functional core of the monitor server. The functional core uses the namespace abstraction to implement core debug services.

### Device Namespace

The Device Namespace should be considered as the progenitor of the namespace model as a whole. It will be remembered that the original intentions of this research was to develop a protocol and debugger model that allowed the debugging of heterogeneous systems within a distributed environment. The problem with heterogeneous environments is that they are different! Even among target systems which have the same type of processor, configurations vary widely with regard to installed subsystems. Of even greater concern is the issue of systems whose hardware configuration can change *dynamically* during the operation of the environment, something that can happen with embedded systems such as telephone exchanges, etc.

The namespace model was initially developed with the aim of capturing the notion of hardware configuration building blocks. The target system was viewed as a group of hardware objects each of which had methods to access and manipulate the object. However, as will be demonstrated in later sections of this chapter, the namespace model has considerably greater general application as the monitor software is actually described as a hierarchy of connected namespace environments.

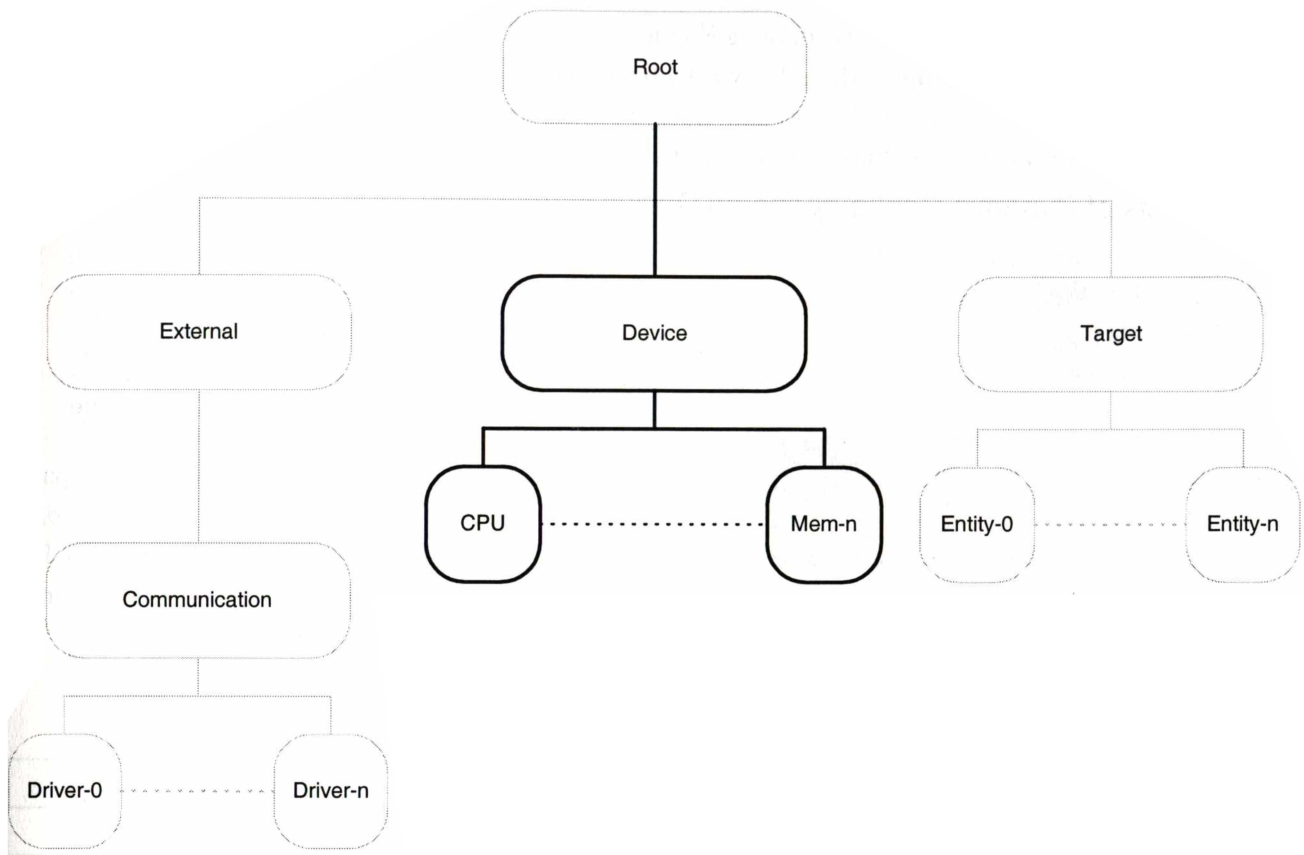



Fig. 5.2 *Device Namespace and its relationship with the Namespace Framework*

The device namespace environment always exists as part of the monitor namespace hierarchy. Device namespace contains *all* other hardware namespace environments. While most namespaces contain a mixture of *Environments, Env-Handlers* and *Work-Handlers*, Device namespace (like Root namespace) is different due to the fact that it only contains namespace environments—in this case, the namespaces which represent the hardware devices that make up a target configuration.

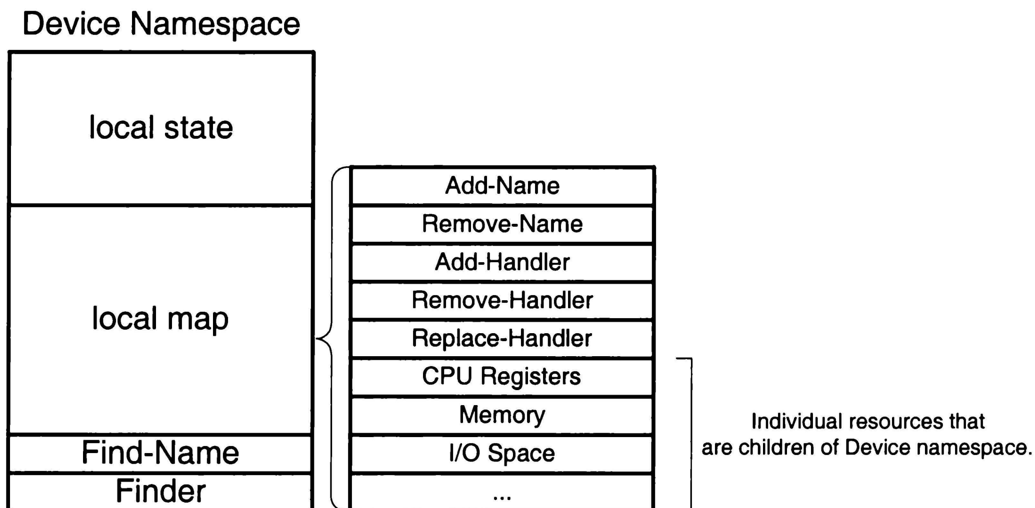



Fig. 5.3 *The Device Namespace hierarchy*

Due to the nature of the namespace model and its base operations, changes in target hardware configuration at start-up or run-time can be viewed as additions to and removals from Device namespace. For example a card being "hot plugged" into the target results in a namespace corresponding to that card being added to that environment. Likewise, the removal of a card will result in the removal of the corresponding namespace. These changes of configuration are always

relayed to the debugger so that the debugger can adapt to changes in the target's configuration. This synchronisation of configurations known to both debug agent and monitor is important as it adds an element of security. A device that does not exist will have no namespace environment, hence access to a non-existent hardware component cannot take place.

The Effects of Dynamic Configuration on Device Namespace

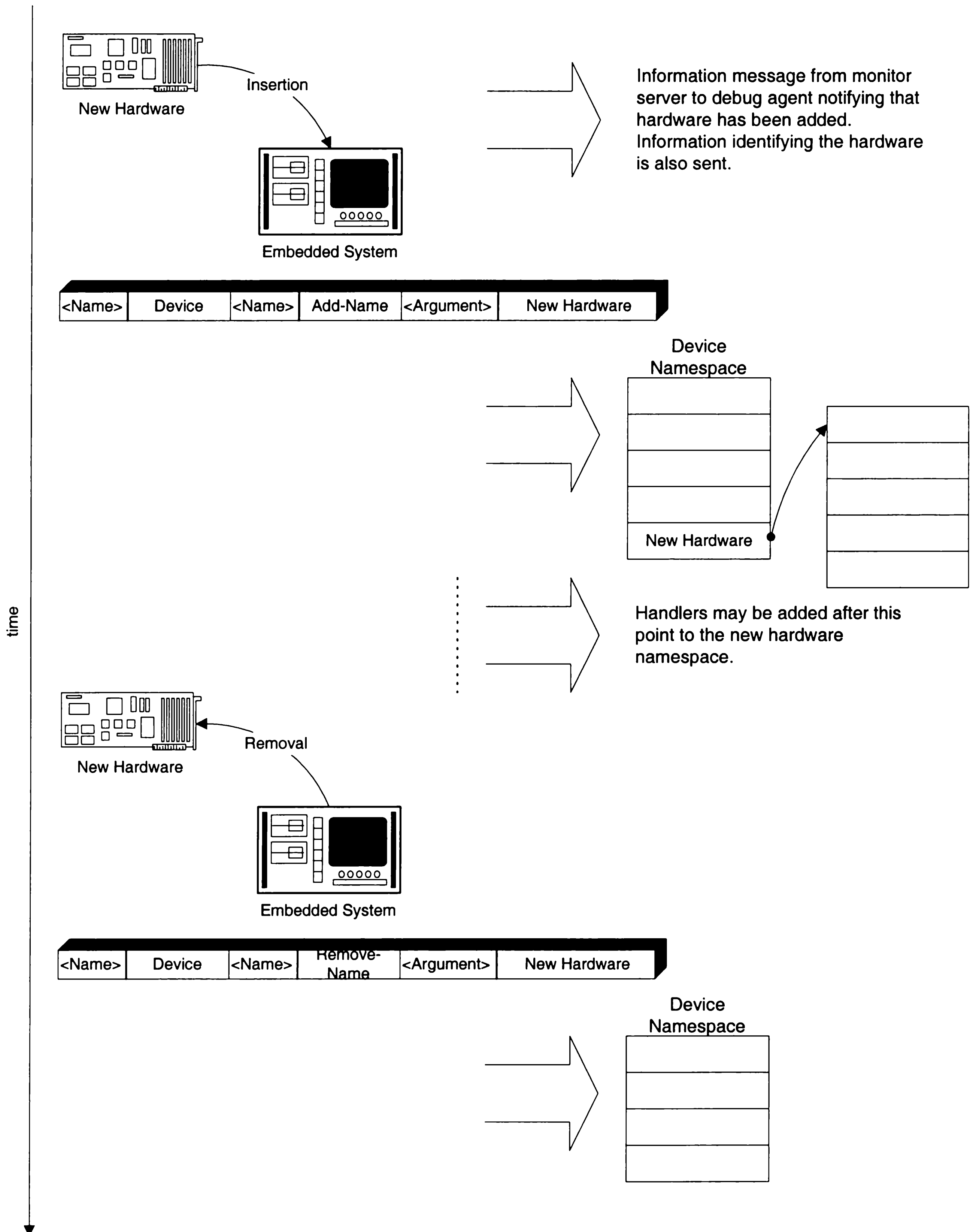



Fig. 5.4 *Namespaces and dynamic hardware configuration*

Configuration of newly inserted devices can take place in two ways:—

(1) If the target hardware provides *interrupt-driven* dynamic configuration then the interrupt handlers can assist in the task of acquiring configuration information and adding or removing namespaces from device namespace.

(2) If no such interrupt-driven configuration is provided, the *debug agent* can send messages to device namespace that add or remove namespaces as configuration changes take place. Some target machines may not support dynamic configuration of any form taking place. Purely static configuration where the target has to be powered down, requires the *debug agent* (not the monitor) to be in charge of the task of adding and removing device namespaces.

The first method of dynamic configuration implies that there are two stages; the process of acquiring configuration information about the device just added or removed followed by addition/removal of the namespace. These actions are performed by the *Dynamic-Add* interrupt handler (this is described in the co-routine handlers in Chapter 6). Two messages are enqueued onto the monitor message queue:—

a. The first message is added to the tail of the queue and contains configuration information relevant to the debug agent.

b. The second message, also added to the tail of the queue, contains instructions to add (or remove) a namespace of a given name from the device namespace.

Most of the work is actually performed by the *Dynamic-Add* and *Dynamic-Remove* interrupt handlers (if the target hardware supports this). Device namespace is a repository for *all* hardware devices present on the target. All hardware devices are modelled as namespaces within Device namespace. Part of the initialisation of Device namespace will be to provide handlers for the "bare" system, that is, the system as it would appear from a cold start.

For each hardware device added to the system there is a map that defines the address space of the device. For example, a memory board will probably be a contiguous block of storage locations that starts from a configured base address to a length that represents the size of the storage block. In other situations, the mapping might be to a sparse address space or may even reside in a totally different memory space—for example, input-output space. This information is useful to both debug agent *and* the device namespace being added. Part of the configuration information given in the first message is memory mapping information in addition to the product information useful for determining symbol descriptors by the debug agent. This is *Port* information, each hardware object (namespace) also contains port information that resides in the namespace state for that particular device.

```
types
    Slot-Id =
            token;

    Port ::
        base : Address
        size : N1;
```

The *Port* structure models hardware that may use the address space to which it is mapped, in a sparse manner. For example, a communications device might map into memory in a manner where each register is followed by a hole in the address space which maps to an unaddressable area. Alternatively, the mapping may represent an input-output address mapping that individual handlers could use to access device registers.

types

*Add-Hardware-Args* ::
*device-name* : *Name*
*hw-info* : *Hardware-Resource*
*ext-env* : *[Env]*;

*Hardware-Resource* ::
*ports* : $Port^+$
*product-Id* : [token]
*manufacturer-Id* : [token]
*unit-serial* : [token]
*comment* : [$\mathsf{char}^+$];

*Device-State* ::
*port-descr* : $Port^+$;

Each hardware resource added has an associated *port-descr* that is used to describe the map of addresses that the resource uses. This port map is determined at configuration time, either by hardware/firmware or possibly the user of the debug agent supplying the address map if the configuration is not dynamic.

state $\rho_{dev}$ of
*dev-ws* : *Workspace*
*dev-map* : *Map*
*my-name* : *Name*
*finder* : *Env-Handler*
init ... – This environment is initialised as part of Monitor global state. Refer to Annex B.
end

## Adding Hardware

The use of *Add-Name* is insufficient to initialise the port information that is required as part of the state of the handler. A specialised *Add-Hardware* handler is required for the Device namespace that uses the *Add-Name* handler and also initialises the port information that the handlers require for manipulating the hardware resource. No specialised *"Remove-Hardware"* routine is needed as *Remove-Name* is all that is required for a hardware device to be removed. Refer to Annex B for details.

*Add-Hardware* consists of two phases, the addition of the namespace which also includes base address information for the device concerned and the determination of whether the hardware resource has a corresponding communications device driver that may be useful to the monitor for switching communications channels during a session.

Device namespace maintains no database of configuration information. This may seem odd given the fact that dynamic configuration may occur often, during and outside a debugging session. This is not a problem, however, as the message queue can be used to provide a *store-forward* mechanism when the debug agent is not connected to a monitor. It is the responsibility of the debug agent to maintain some overall configuration for the target machine to which it connects, the target merely notifies of any changes so that the debug agent can keep its "view" synchronised. Details of the exact mechanism involved are given in the definition of the *Interrupt-Coroutine* in Chapter 6.

In the case of interrupt initiated dynamic configuration, messages are queued to the monitor as they occur—even when there is no active session between debug agent and monitor.

NOTE The monitor functional core adapter contains mechanisms that queue messages and synchronous/asynchronous events to be processed by the namespace engine. The messaging primitives are described in detail in Chapter 6.

When the monitor becomes active through a session being established, all configuration messages that have been enqueued are processed by the monitor with any relevant replies being passed onto the debug agent.

When an interrupt-driven *Dynamic-Add* occurs, the same sequence of messages is enqueued. If the namespace addition was successful then the pairing of descriptor and namespace takes place. The debug agent now has a complete description of the target configuration.

The enqueued messages are only acted upon when the monitor is executing and processing messages from the queue. An earlier iteration of the design had the *Dynamic-Add* interrupt handler using *Push* to place the the first message at the head of the queue, with the second message being placed at the tail of the queue. This solution unfortunately had race problems due to the primitive queue operation *Push* upsetting the temporal order of events enqueued. It was quite possible to have a Device namespace remove a message *before* the addition of the device due to the *Push* operation! Queuing the two messages in a strictly FIFO manner prevents such situations from occurring.

### Typical Device Work-Handlers

Within each device space there will be typically one or more *Work-Handlers* that provide an abstract interface to the device. For devices, the interface will provide *Put* and *Get* operations that act upon the resource. For a register file, there will be appropriate *Put-Register* and *Get-Register* operations. Ultimately, it is *Work-Handlers* which "characterise" a particular namespace, separating the functionality of one device from another. *Work-Handlers* for a particular device exist either at start-up (if they are part of the motherboard target configuration) or get added, removed or replaced at run-time as devices are added or removed.

What is a suitable interface for *Put* and *Get* operations? An *offset*, *offset-value* or *offset-value-count* is useful for such operations. How the operations are ultimately performed is immaterial, actual handler implementations map the values given above into concrete values of the target device. The definitions of *Put* and *Get* are ultimately implemented as *retrieval functions* for a given target implementation. Refer to Annex B for details.

As interface definitions for each handler are known by the debug agent there is no need to check for the correctness of parameters that are given as part of the message to the handler when it is invoked as a service by the debug agent. It is the responsibility of the debug agent to correctly construct messages in the right form for use by the respective handlers that are *not* part of the model, that is, handlers and namespaces that are outside the set of default debug services described in this chapter.

## CPU Namespace—A Namespace within Device Space

The typical target system consists of similar kinds of hardware components as do most computer systems. The target system's memory, input-output and CPU registers are modelled by namespaces within Device namespace. Memory and input-output devices are directly accessible using the handlers of such namespaces. The state of such hardware is directly manipulable from the handlers by effectively reading from or writing to an address of an object managed by that namespace.

The handlers of CPU namespace have a more difficult task. The CPU namespace represents the register file of the entity that was last interrupted, not the current contents of the registers

as the monitor executes. This means that the CPU namespace must have access to the context of the entity that was running *before* the entity was stopped. This context is different for each entity. Each entity in existence has a context area containing a register dump area representing the register state of the PUT before the interruption occurred. As a result, the pointer to that context will change depending on which entity was active at the time of interruption.

Given a handle to the last interrupted entity's context, CPU namespace can access and manipulate the register file in the same way the memory and input-output namespaces manipulate their respective namespaces. The original CPU namespace is static. Additional CPUs may be added but the original cannot be removed before there is at least one other CPU available.

*CPU-State* ::
  *ports* : $Port^+$
  *register-dump* : *Address;*

state $\rho_{cpu}$ of
  *cpu-ws* : *Workspace*
  *cpu-map* : *Map*
  *my-name* : *Name*
  *finder* : *Env-Handler*
  init ... – This environment is initialised as part of Monitor global state. Refer to Annex B.
end

The *register-dump* state refers to an area *within* the workspace of the PUT used to maintain the saved registers of the suspended PUT. The *register-dump* state is initialised by Root namespace because it has access to the CPU namespace state. The *register-dump* value is analogous to the *Port* structures used to describe the mapping of hardware devices and is initialised to point to the saved register file of the PUT each time a transfer of control takes place from PUT to monitor co-routine by the trap co-routine. This will be discussed further in Chapter 6 in the section on monitor initialisation and Root namespace.

### Memory Namespace

Unlike the CPU namespace, devices such as memory do not have the same problems of maintaining the last activation in order to access memory. The entity context information required by CPU namespace is due to the possibility of multiple entity activations being present when an entity shares the same resource (registers) with other entities. There is only ever one *physical memory*, how this memory is partitioned by the PUT is immaterial as handlers only deal with *physical addresses*. Handlers similar to the *Put* and *Get* handlers exist for each hardware resource whether they are in memory or input-output space.

## Target Namespace

The ability to control the execution and debugging of the PUT is the responsibility of Target namespace. Logically, Target namespace consists of three components:—

(1) Control—the functionality to control the execution of the PUT. This provides the essential ability to commence or recommence the execution of the PUT.

(2) Entities—are namespaces subordinate to Target namespace that are used to identify the activity executing at the time a break or trap event occurs.

(3) Breakpoints—software support of a target specific means of halting the execution of the PUT at a specific point in the control flow of the PUT.

Support for breakpoints is provided by Work-Handlers within Target namespace that allow the debug agent to set and clear breakpoints and allow the monitor to determine *what* breakpoint has occurred and *which* entity was associated with that breakpoint. Figure 5.5 illustrates the relationship between target, breakpoint, program control and entity namespace:—

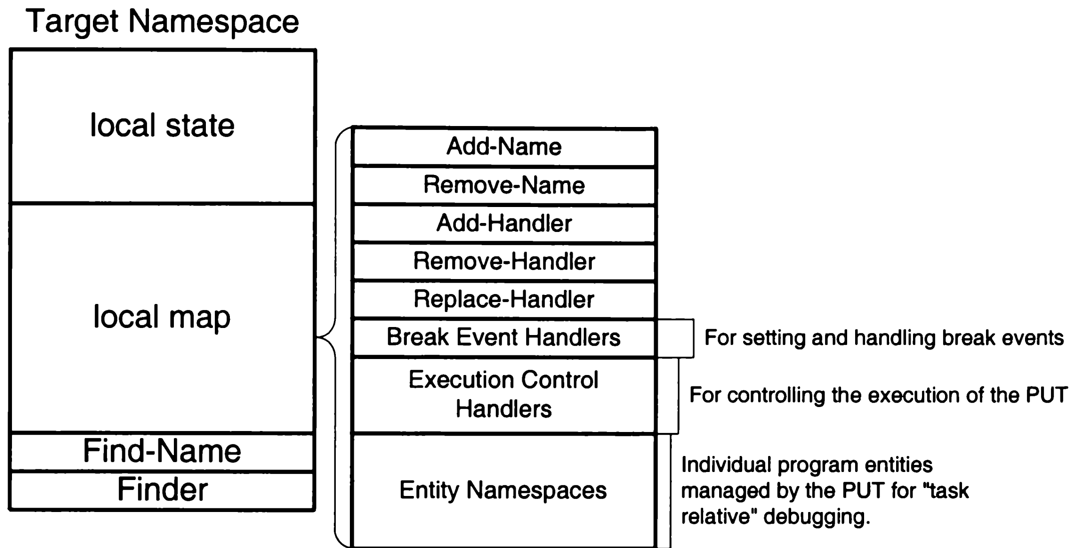



Fig. 5.5 *Target Namespace contains Entity and Breakpoint functionality*

Device namespace was described as managing hardware resources; Target namespace should be considered as a manager of software objects (entities). The analogies go further than this. The Target namespace is subject to dynamic addition and removal of entities as the PUT adds or removes entities such as processes, co-routines or objects throughout the life of the PUT. In principle, this is exactly the same as what happens with the dynamic hardware configuration and Device Namespace.

Target namespace also provides methods for creating a program context necessary to start the PUT running. In an earlier iteration of the design, such handlers were considered to be a part of another namespace called *Control namespace.* This was found to be unnecessary as the PUT is initialised in a manner almost identical to registering an entity (except that it is done before the program starts), hence program control also is part of Target namespace in addition to breakpoint *Work-Handlers* and Entity namespaces. Much like the *logical* breakpoint and entity namespaces, control namespace will be treated in the same manner. The Control namespace supplies three handlers, *Initialise-Program, Set-Finalisation* and *Continue-Program.* These handlers are sufficient to allow the commencement and continuation of the PUT.

The state of the Target namespace is composed of the breakpoint table used for breakpoint support and the PUT co-routine. The PUT is treated separately from the other monitor co-routines because it has its context manipulated by handlers which control its execution.

*Target-State ::*
*put-ws-base : Address*
*put-ws-size :* **N**
*put-pc-ptr : Address*
*put-finalisation : [Address]*
*saved-id : [Break-Identity]*
*program : Coroutine*
*program-under-test : Context*
*breaks : Break*
*code-map : Code-Map*
*run-state : Exec-State;*

The PUT is an example of an *entity*, an independent thread of control. The PUT may subdivide itself into a number of independent entities—they could be tasks, processes or co-routines. These may be *registered* by the program so that the human user can manage the complexity of debugging a concurrent program more effectively using the debug agent. Entities established by the PUT are *subordinate* to the PUT, and therefore entity namespaces are subordinate to Target namespace.

The values of the first three elements of Target namespace state are established when the PUT is loaded. The workspace of the PUT is *inclusive* of the workspaces of each of the subordinate entities. Target namespace *is* the PUT entity. The first two elements of the state are identically laid out to the state definition of the Entity namespaces discussed in the next section. This is organised in such a manner that the *Find-Entity* Env-Handler can determine if an event occurred during the execution of the PUT or the monitor.

state $\rho_{tgt}$ of
  *tgt-ws : Workspace*
  *tgt-map : Map*
  *my-name : Name*
  *finder : Env-Handler*

  init ... – This environment is initialised as part of Monitor global state. Refer to Annex B.
end

NOTE The PUT entity is present *before* the PUT executes. The program entities exist as entity namespaces only *during* the execution of the program.

The *Exec-State* type is used to describe the current execution state of the PUT. Abnormal termination through traps and breakpoints causes the PUT to go from the RUNNING state to the STOPPED state, successful termination causes the PUT to go from the RUNNING state to the TERMINATED state. The initialisation of the PUT co-routine causes the execution state to go from either TERMINATED or NO-PUT to LOADED. The commencement of the PUT causes a state change from either LOADED or STOPPED to RUNNING. The state TERMINATING is distinct from the state TERMINATED. The TERMINATING state is only entered when global breaks occur. This causes each entity's finalisation code to execute. If TERMINATING entities *complete* their finalisation and the program terminates then the program execution state is set to TERMINATED.

types

*Exec-State* =
  ⟨*NO-PUT*⟩ |
  ⟨*LOADED*⟩ |
  ⟨*RUNNING*⟩ |
  ⟨*TERMINATING*⟩ |
  ⟨*STOPPED*⟩ |
  ⟨*TERMINATED*⟩ ;

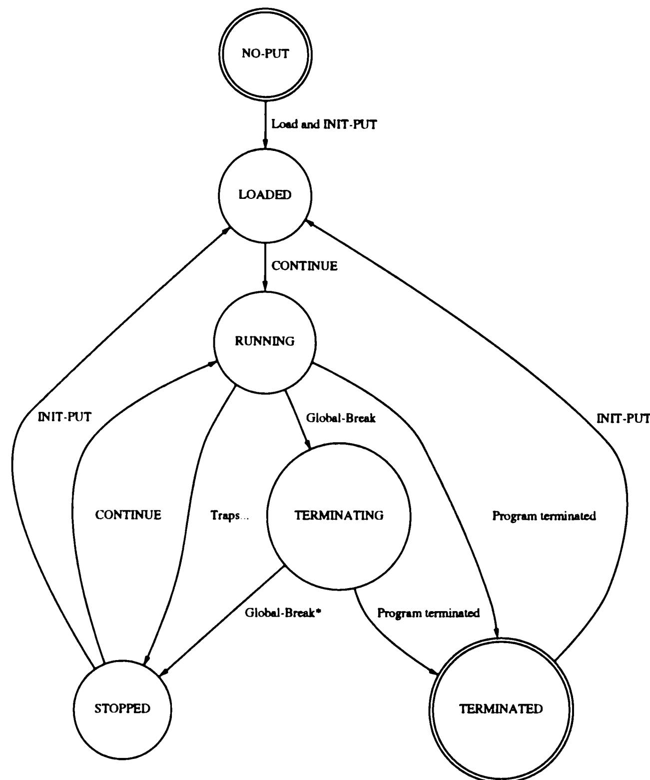



Fig. 5.6 *PUT—execution state diagram*

* The directed arc that emanates from the *TERMINATING* state to the *STOPPED* state is issued by the debug agent to halt the PUT where there is the *possibility* that the finalisation has not resulted in termination of the PUT.

The reason for having a TERMINATING state is to avoid deadlocks or livelocks occurring due to finalisation not causing program termination. This can mean that potentially the monitor may *never* regain control. This is solved by being able to *abort* finalisation by having the debug agent send a global break message *during* finalisation. This terminates the PUT and forces control to return to the monitor. Because such termination is ungraceful, there is no guarantee that the PUT has returned any of the hardware to a *safe* system state.

## Entities

One of the most difficult issues of debugging concurrent programs is that of making sensible inferences of a PUT which consists of multiple threads. In general, a single threaded program is easier (though not trivial!) to debug because there is only ever one activation that executes the code. Because of this, fault localisation is considerably easier. For a software system that uses multi-threading, knowing *where* a program fails is not sufficient. If a number of threads share a particular code section, say as part of some library module, there needs to be a way of identifying *which* thread was active when a breakpoint occurred in addition to *where.* Not supporting such functionality in an embedded systems debugger increases the difficulty of locating faults or capturing the moment when a thread exhibits erroneous behaviour or absolute failure.

Entities allow the determination of *which* thread was active at the time a trap or breakpoint occurred. As a result of this, debugging can take place at a considerably higher level of process abstraction because code executed can be associated with a particular thread of control.

To explain how entities work requires a brief examination of what entities actually represent. An entity is an abstract description of a flow of control, a thread, a co-routine, a process or possibly some other active data object. These objects are dynamically created or destroyed during the run-time of the PUT. *All* entities have their own *Workspace*, for local use which is mutually disjoint

from all other entities. The fact that each entity has its own workspace is used by the Target namespace to determine *which* entity is active. When a trap occurs while the PUT is executing its context is saved and the trap handler determines the reason for the trap in addition to some values such as where the interruption took place and the current stack pointer. The current stack pointer can be used to determine the particular entity by bounds checking the Workspace against the current stack pointer. If the stack pointer lies *within* the Workspace then that is considered to represent the entity active at the time the trap occurred, otherwise it was some other entity.

If the PUT wishes to make use of the entity facility, it is responsible for *registration* and *removal.* The PUT invokes the registration and removal of entities through services provided by the *User Services* "library". These routines construct messages that are enqueued to the monitor for processing the next time the monitor becomes *active.* The messages enqueued are service invocations to add and remove entities within Target namespace. The overhead is only that of enqueuing a message to the monitor, not the *processing* of that message by the monitor. The intrusiveness of registration and deregistration of program entities therefore has minimal performance impact on the PUT, considerably less than the impact breakpoints have on the performance of the PUT. The processing (adding or removing entity namespaces) is performed when the PUT is *not* active. The issues of breakpoint intrusiveness will be discussed in a later section.

Entities provide the *monitor* and the user of the debug agent a method of attaching meaning to a complex concurrent program. The PUT does not have to register every entity, it may only register those entities which are prone to failure or erratic behaviour. In conjunction with breakpoints this ability can be used to isolate *which* entity actually caused the fault. A logical argument could be constructed by the human user of the debug agent along the lines of, "If this trap did not occur in the entity I registered but in something I hadn't registered then the entity I registered is not the entity at fault."

The *User Services* library provides two operations that register and remove entities from within target namespace. These operations are invoked synchronously by the PUT but their actioning is *asynchronous.* The operations provided for registration and deregistration of entities amount to queuing messages. This is the key to why entities cause no perturbance of performance of the PUT. The *actual* registration or removal is actioned only while the monitor is *active.* For this to happen, some event must cause the PUT to be trapped thus forcing the monitor to be resumed. What could happen in this case is a deliberate means of causing the trap to occur just before the main program commences. This would allow any entities to be registered by the monitor (or possibly even removed). In this state the user of the debugger could use the entity information (the name of the entity) returned by the monitor and associate the entity with a particular breakpoint. When all breakpoints required for debugging have been set, the PUT could be continued from the point where the trap occurred so that testing can then take place.

Entities take advantage of the fact that active data objects have local state associated with their execution, a *workspace* used by the entity possibly for maintaining local variables, procedure activations, module state and thread context. To the monitor, entities are treated as names within Target namespace. The PUT as described in the state definition for Target namespace is also an entity, its namespace is *equivalent* to Target namespace. For an entity to be located, all that needs to be known is the base address and size of the workspace that the entity uses. This information is provided at *run-time* by the *PUT* that registers the entity.

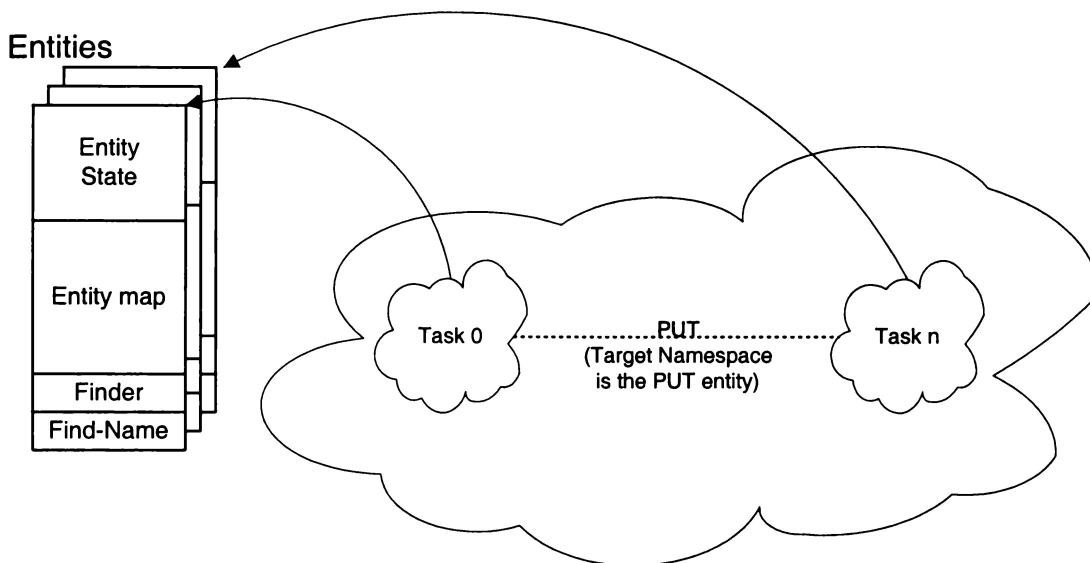



Fig. 5.7 *Entities are an abstraction of object execution control flow*

An entity may actually represent a variety of active objects: processes and co-routines are good examples. What the entities actually are and how they are implemented is of no concern to the monitor. If an entity does actually represent a process or co-routine then that entity has a closure and program counter of its own. The entity descriptor also has a pointer to the *program counter register* within the context of the entity as well as a pointer to the *finalisation code* of that entity. The need to maintain a reference to the program counter within an entity closure is used for specifying group breakpoints. The finalisation code entry point is used by the global breakpoint mechanism to ensure *safe* breakpoints when all entities are brought to a stop on all target machines under test. The usage of these will be explained in more detail in the section describing breakpoint handling.

NOTE Finalisation *may* be established for entities. The nature of such finalisation is dependent on both the run-time organisation of the PUT with respect to the establishment of the entities and the language support provided.

The entity model uses *Add-Handler* and *Replace-Handler* to allow breakpoint predicates to be associated with a given breakpoint and entity. The name of the predicate is the breakpoint descriptor. When breakpoints occur, the predicate for the entity active at the time the breakpoint occurred is executed to see if some assertion was satisfied. If the predicate returns true, the PUT is halted.

As the PUT co-routine is an entity in the same manner as subordinate entities, predicates may also be added as *Work-Handlers* within Target namespace. A reasonable interpretation of this is that the PUT entity is effectively the parent entity of all subordinate entities with only the extra state and handlers of what is actually target namespace being indicative of the extra capabilities that need to be performed. From a structural point of view, Target namespace and its subordinate entity namespaces are the same—the PUT co-routine has a name (Target namespace) and a workspace base and size used to maintain stack and program context just like entities do. The PUT entity also has an associated finalisation entry point and current *program-counter* position.

types

*Termination* =

$() \rightarrow [\mathsf{token}];$

*Entity-State* ::
*base* : *Address*
*size* : $\mathbf{N}_1$
*pc-ptr* : *Address*
*finalisation* : *Address*
*saved-id* : *[Break-Identity]*

NOTE The entity state contains the *saved-id* element for use when *delayed* break events are being established when an external group break event has occurred. The use of the *saved-id* element will be described in relation to group break event handling in the sections describing breakpoint event handling.

There are three operations for manipulating entities (refer to Annex B). All are examples of *Env-Handlers*:—

(1) *Add-Entity*—a specialised *Add-Name* used to register entities within Target namespace and also initialise the state of the entity, the base address and size of the entity's workspace. A pointer to the entity's current code position as well as the entity's finalisation code entry point are set for use by the breakpoint model when establishing group or global breakpoints.

(2) *Remove-Entity*—a specialised *Remove-Name* used to deregister entities within Target namespace. The removal of an entity is complicated by the use of entities by the breakpoint handlers within Target namespace. The removal of an entity removes its representative namespace but also removes entries in entity sets in breakpoint descriptors. A useful side effect of having predicates as part of an entity namespace is that as soon as the entity is removed, the predicates associated with breakpoints set on that entity are also removed. Predicates are removed from within that entity before removing the namespace. If the entity group of a particular break entry becomes empty, that break entry is removed as breaks have to be set with at least one entity in the set. The break event mapping is described in more detail later in this chapter.

(3) *Find-Entity*—determines the entity active when the trap or breakpoint occurred. *Find-Entity* works by isolating the name environments within target namespace and doing a bounds test on the workspace descriptor. The *Find-Entity* operation takes a *Trap-Msg* as its argument and uses the current stack pointer information of the interrupted PUT context to do the bounds test. The name found is returned.

In the case of *Find-Name* there will always be a name returned, if it is not one of the entities registered, then it will be the original co-routine representing the PUT as loaded by the monitor. If it is not the PUT co-routine then it is a *user defined Work-Handler* within the monitor itself that caused the error! This means that the monitor can be used to debug itself when required.

## Controlling the Program Under Test

To the monitor, the PUT consists of code and data. The user program is a co-routine, the monitor controls the execution of the PUT co-routine to accomplish any debugging. To execute the PUT requires that the following steps be taken:—

(1) Downloading the PUT to a particular target requires actually writing code to Memory namespace, a task that can be easily accomplished by using the *Put* handler within Memory namespace.

(2) Activation of the PUT requires a co-routine context to exist and be appropriately initialised.

(3) Initialise the finalisation code of the PUT if required.

(4) Finally, continue (or commence) the execution of the PUT.

There is only ever *one* PUT while there may be many entities within the PUT, seen as a co-routine in relation to the monitor. The PUT is at liberty to sub-divide itself by any means required for the job to be performed but the monitor only "sees" one co-routine.

Establishing the PUT from the model's point of view requires the establishment of the entity that represents the PUT. This means that the workspace *base* and *size* must be established in addition to the locality information (stack and program counter) required to start the PUT. The *run-state* is also changed to LOADED.

From an implementation perspective, establishing the PUT is more involved. The co-routine representing the PUT has to have the context which is part of the co-routine initialised. This means the establishment of the closure such as the register file and the construction of a call stack. The entry point at which the PUT commences execution has to be established by setting the program counter, local variable and procedure call information is maintained by the stack within the workspace. All these structures are a part of the context of the PUT. This context is contained by the workspace for that co-routine.

Termination code must also be established for when the PUT completes. If the PUT terminates abnormally, the trap handlers defined in Chapter 6 will ensure that control is eventually returned to the monitor. For the case where the PUT terminates normally, control has to be returned to the monitor. This model *termination* code is separate from the case described later in the breakpoints section which defines *finalisation code* for each entity. Termination code allows the graceful return of control from the PUT to the Monitor co-routine. In a successful termination, the termination code runs *after* finalisation code of each entity has taken place.

One method that can be used to ensure that the model termination code is run when normal, *successful* termination occurs is to take advantage of the call stack which is part of the workspace of the PUT co-routine. Such termination code could be invoked by specifying a *return address* at the top of the stack that causes the termination code to run. The termination code performs the following steps:—

a. Set the *run-state* to TERMINATED indicating that the PUT has actually terminated successfully of its own accord.

b. Construct and enqueue a message to the monitor describing the reason for termination. Unlike the reasons for abnormal termination which are ERROR messages enqueued to the monitor, normal termination is a RESPONSE sent back to the debug agent.

c. The last operation performed by the termination code is to transfer back to the monitor. The monitor will transfer control to the point at which it transferred to the PUT, probably the result of the *Continue* handler being invoked.

values
*put-reg-dump-base : Address* = undefined – implementation-defined

Annotate

This value is the address of the program-under-test co-routine's *raw saved context.* This address and successive locations in the register dump is necessarily *within* the workspace of the program-under-test co-routine.

End Annotation

values
*put-pc-dump-base : Address* = undefined – implementation-defined

Annotate

This value is the address within the raw saved context of the program-under-test co-routine's saved *program-counter*. The saved program counter at this address may be altered by the break handler to establish finalisation or delayed break events. The address value is within the workspace of the program-under-test co-routine.

End Annotation

NOTE For specifications on each of the following control handlers of Target namespace refer to Annex B.

(1) The *Initialise-Program* handler is a *specialised* form of the *Add-Entity* presented earlier—it is for this reason that the handler has an *Env-Handler* signature rather than the (expected) *Work-Handler* signature. The purpose of this handler is to establish the PUT *entity*, the program context has to be initialised and all *previous* entities and break mappings removed. *All* break events in both the *breaks* table and *code-map* table have to be cleared. If this were not done the break events used by a previous execution could interfere with those of a new (or possibly modified) program about to be executed. *All* entities associated with the last execution will also be removed before the new PUT is started.

(2) *Establish-Finalisation* is used to establish any "special" *finalisation* code that should be executed by the PUT to ensure a safe shutdown and return to the monitor. Typically this is given the nil value implying that there is no finalisation to be executed. Program termination will *always* be executed during a normal PUT termination, finalisation is optional.

(3) The *Continue-Program* handler takes a loaded program previously initialised by *Initialise-Program* and transfers to the PUT. If the *run-state* is other than LOADED or STOPPED, the PUT cannot be continued.

## User Services

User services are not modelled as a separate namespace but as a small group of *publicly available library routines* enabling the PUT to invoke certain monitor namespace handlers.

User services are provided to support the PUT in two ways. The first method is not so much provided as a service for the program-under-test but the debug agent. The registration and removal of entities is used to support a "thread" level view of computation used by the breakpoints model for selective fault location. The second facility provides primitive input-output using the channel between debug agent and monitor. This is equivalent to the simple *get-char* and *put-char* services typically provided by monitors for system development.

The operations provided to register (*Register-Entity* and *Destroy-Entity*) and deregister entities do not perform the actual operation, instead they construct a message and enqueue it for future processing by the monitor when it becomes active. This means that there is a delay between the actual registration of an entity as part of the execution of the PUT and its corresponding recognition by the monitor with subsequent addition or removal of namespaces from Target namespace.

The *Register-Entity* and *Destroy-Entity* user services are provided for use *only* by the PUT, refer to Annex B for details.

An entity is characterised by the workspace used to maintain its state. This state area is used for many purposes, local variables and closure information being two of them. Entities are self contained and maintain enough information for an independent existence—in a multi-threaded environment, this information is used to maintain sufficient state to allow resumption of an entity.

This continuation state is maintained *only* as part of the workspace of the entity. Therefore *pc-ptr* must be an address that is within the locality of that entity. If it is not then that entity is invalid and does not get registered. This behaviour is required by the model for the purposes of the breakpoint handler, particularly the group break handler which may have to set *delayed* breaks based upon the interrupted entity's current program counter .

User services also provide primitive input-output to the PUT. This can be used to provide a debug input-output service between monitor and debug agent. The service is *message block* oriented, it is up to the PUT as to how to use the message block.

The *User-Get* (refer to Annex B) operation returns a message to the PUT. The operation returns immediately if there is a message in the one message buffer, otherwise it blocks on the idler waiting for a user data message to arrive.

The *User-Put* (refer to Annex B) operation is somewhat more straightforward due to the fact that neither buffering nor blocking is done. The message send operation is *synchronous.*

The user service input-output operations are only used by the PUT for message passing between debug agent and monitor. The services could be extended to handle different sources but this is of debatable use. If the PUT wants to pass messages between other monitors it should do so using it's own device drivers and not the monitor's. This is in keeping with the "Monitor—not Operating System" philosophy of the design. Arguably, the operation of *User-Get* is not ideally suited for use by the PUT as a means of receiving messages from and sending messages to other "parts" of the PUT executing on other monitors. Making the user service input-output more flexible may also have the detrimental effect of increasing the monitor's intrusiveness in the PUT.

## Breakpoints

The breakpoint is the only means the debug agent has of controlling PUT execution. Most debugging activity is associated with examining the state of the program when frozen—a state due to either a trap occurring or a user specified breakpoint being encountered.

The traditional breakpoint that most programmers have experienced using as part of the activity of using a debugger to debug a software system is given the name *local breakpoint* in this thesis.

> *An event defined by a local breakpoint is said to occur when some target-dependent occurrence, such as a trap, takes place.*

The occurrence of such a break event does not imply that a halt in the PUT has occurred—merely that tests specified in defining the breakpoint must be used to indicate if and where a break occurred. If the event did not define a break then the PUT is resumed. If it did then a break event notification is sent to the debug agent.

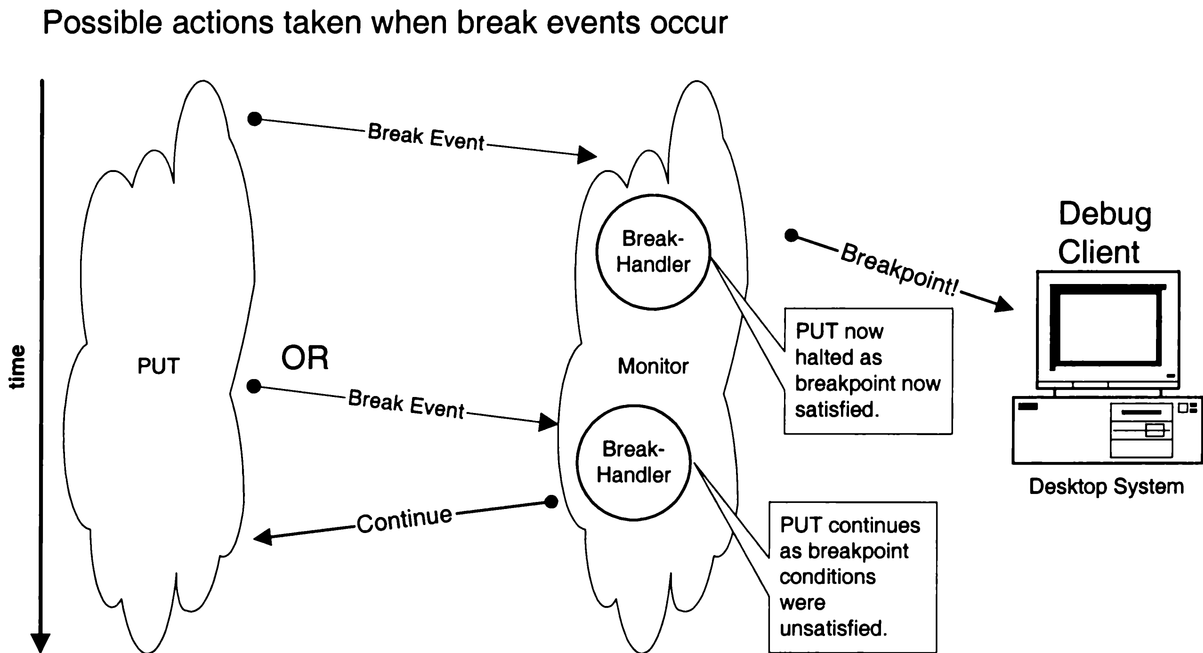



Fig. 5.8 *Break events may have an effect on debugging dialogue*

A *watchpoint*, another form of breakpoint, may halt execution of a program when a read or write to a single location or block of memory locations occurs. This thesis uses *predicated breakpoints* to emulate the behaviour of a watchpoint.

Debugging concurrent programs is made considerably more difficult due to the problems of identifying a thread associated with the current program counter. In a program that implements its functionality through *multiprocessing* many threads could be executing the same shared code section. For systems which fail in a shared code section it is extremely difficult to localise the erroneous thread. The breakpoint mechanism described by this thesis uses *entities* (previously described) to reduce complexity when debugging large, concurrent and possibly distributed threads of control.

The significance of the entity is unknown to the debug agent and monitor. The monitor adds or removes entities within Target namespace. Entities have a limited number of operations that can be performed on them. They can be added or removed by name, have predicate handlers added or removed and have their workspace area range checked.

The usefulness of entities is only realised by the extra expressibility that an *entity relative* breakpoint allows. A breakpoint can be taken, or ignored if an entity *not associated* with the current program counter is active, when the break event occurs. This extra discriminatory power makes entities useful to the breakpoint model, particularly with concurrency in mind. The combination of entities and breakpoints then provides a higher level *task oriented* view of the behaviour of the PUT.

The breakpoint model defined here allows three additional kinds of breakpoint useful in the debugging of distributed systems:—

a. A group break event may be associated with a group of entities (possibly executing on more than one target). For each individual entity in the group a program counter and predicate must be defined. There is no requirement for these to be the same or different for each entity in the group. A group break is defined to occur when any one entity break is detected. Detection of a group break involves sending a *group break message* to all targets executing entities in the group, in addition to sending a break message to the debug agent.

b. The delayed break event represents a special class of group breakpoint. Delayed break events are established by a *group-break-handler* if the entities associated with that group identity were not executing at the time the group break event arrived. Delayed break events only exist for as long as they take to be satisfied. As soon as delayed break events occur, they are removed.

c. A global break event is associated with all *registered* entities in the PUT. A global break is defined to occur when it is detected within any PUT entity. Detection of a global break involves sending a *global break message* to all other targets involved in the PUT. This is performed by the debug agent on behalf of the original target in which the break event occurred. A global break message is sent to the debug agent and to complete its finalisation the PUT is resumed.

A global break event may be set to occur by setting an appropriate break on a particular monitor or by having the debug agent send a global break message to halt just that target the message is sent to. This provides the debug agent with a means of stopping a target without actually *setting* a breakpoint to occur on that target. This is particularly useful in halting programs that have run amok.

## Break Event Modelling

The incorporation of entities into the breakpoint model necessitates extra processing when a break event is detected. A breakpoint is *always* going to be intrusive. Without the introduction of entities, a break event will *always* bring the PUT to a stop. An entity relative break event is *less* intrusive, since the PUT will only halt when a break event occurs while a particular entity is active. If a break event occurs when an entity *not* associated with that break event is executing, the PUT can be continued. Therefore, an entity based break event provides finer granularity when debugging complex code structures.

The approach taken by the break event model is entity based. A break event may be able to be associated with *any* entity. Therefore, each entity can be associated with a number of break events. The setting of break events is the responsibility of the debug agent. The debug agent provides an entity name used to associate an entity with a particularly break event. This implies that break events can be established relative to an entity or group of entities.

The break event model could have been designed in five different ways in conjunction with the entities mechanism:—

(1) *One break event per entity*–this is too inflexible for the effort required to implement it and results in too much duplication of data across entities.

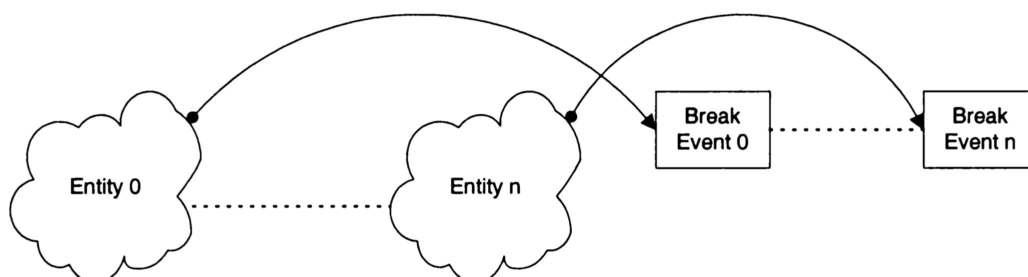



Fig. 5.9 *One break event per entity*

(2) *One entity per break event*–this permits a simple specification and implementation but is unable to associate a group of entities with one break event.

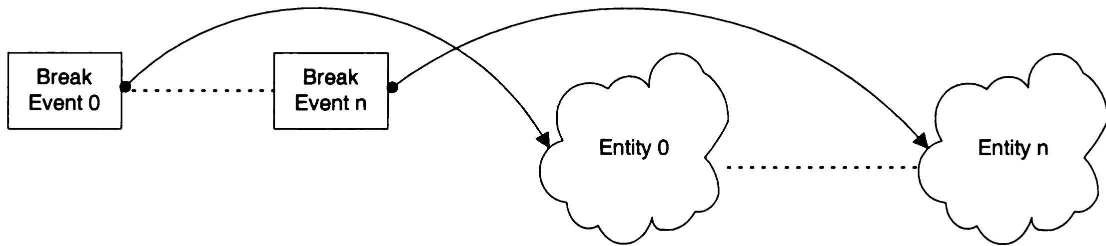



Fig. 5.10 *One entity per break event*

(3) *Many entities per break event*–this is not much more complicated than (2) but allows entities to be associated with a breakpoint. The entities associated with the break event can be added or removed as required.

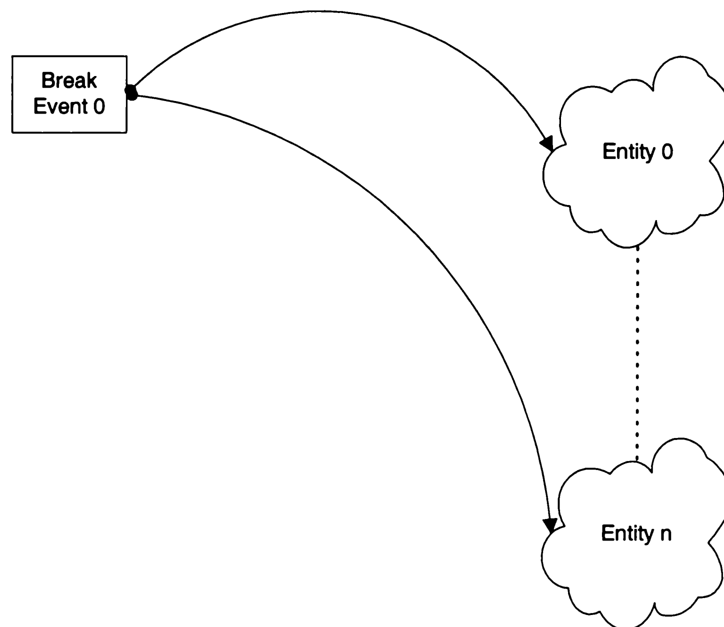



Fig. 5.11 *Multiple entities per break event*

(4) *Many break events per entity*–this solution is useful but cannot take advantage of shared information as (3) does. The implementation cost is high since each entity has to provide handlers for the breakpoint and breakpoint table. The solution does have the appeal that when an entity is removed by the system, the breakpoints associated with it are also removed. This means that breakpoints cannot be set on non-existent entities. Unfortunately, while this association between the entity and the breakpoint set makes sense, there is a problem regarding the setting and clearing of breakpoints that share the same address but are from different entities. The problem is to do with having breakpoints that share the *same* address across entity boundaries. This sharing requires mapping from address to break event which complicates the break event structure and makes the determination of breakpoint instruction replacement/restoration (if breakpoints are implemented by the use of special instructions) extremely difficult.

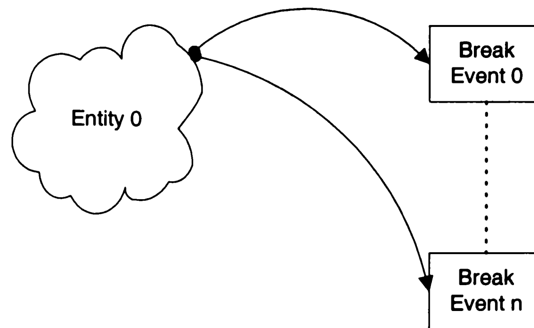



Fig. 5.12 *Multiple break events per entity*

(5) This is a combination of (3) and (4). It is much like solution (3) except that an address may have associated with it several breakpoint identities. A breakpoint identity may have several breakpoints associated with it all at different addresses. Like solution (4) it allows entities to maintain a predicate per breakpoint identity. Predicates are *Work-Handlers*—a truth result would be a SUCCESS while false would result in a FAILURE. When entities are deleted, the predicates associated with that entity are also deleted. However, the additional overhead of deleting entity references from the break event descriptors is incurred.

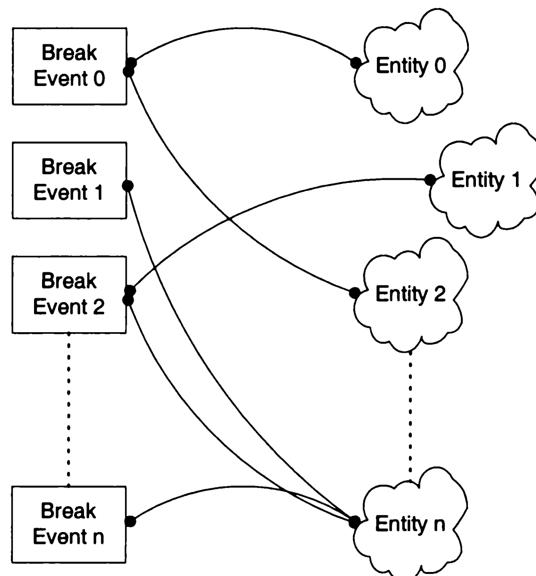



Fig. 5.13 *A many-many mapping of break events and entities*

The solution suggested by (5) has been chosen because it minimises the state by allowing break events to be associated with several entities. As entities are removed from the PUT, any break event associated with that entity *must* have the relevant *entity-break* association removed from that break event table entry. When there are no entities associated with the breakpoint entry then that break event entry itself is *removed.* Depending on how breakpoints are implemented, any target-dependent method that was used to set the break event can then be used to clear the breakpoint so that further break events do not occur even after the breakpoint entry has been removed.

To define global or group break events so that the correct message sending can take place, *all* target monitors must use the same communications medium to co-ordinate break event handling. This can only be enforced by the debug agent. If a monitor is not listening on a particular channel which another monitor considers to be the current channel in use, group breakpoint messages will be lost. The only means of solving this problem is to have the debug agent do any necessary message routeing as the debug agent always knows which channels are in use for each monitor. The approach taken by the breakpoint handler defined in this model is to have the monitor send

group breakpoint messages directly to other monitors involved in that group breakpoint. Global breakpoints at one target result in a reply to the debug agent which then sends a message to all target systems—the debug agent is responsible for message routeing in this case.

types
*Predicate* =
*Work-Handler;*

The predicate for an entity is given a break count and an address which the predicate may use as part of the predicate expression.

types
*Machine-Address* =
token*;*

The *Machine-Address* is a *device-dependent* value that represents the machine network address. It is used by the group break event handler when messaging other monitor servers that are members of a group break event specification.

types
*Entity* =
*Name;*

*Breakpoint-Kind* =
⟨*LOCAL*⟩ |
⟨*GROUP*⟩ |
⟨*DELAYED*⟩ |
⟨*GLOBAL*⟩;

The breakpoint kinds LOCAL, GROUP, DELAYED and GLOBAL denote breakpoints of each type as described earlier.

types
*Break-Acknowledge* ::
⟨*LOCAL-ACK*⟩ |
⟨*GROUP-ACK*⟩ |
⟨*GLOBAL-ACK*⟩;

The *Break-Acknowledge* type is used to provide the debug agent with information regarding the reason for the break occurring. This is particularly pertinent with regard to group and global breaks. The occurrence of a group break causes an appropriate acknowledgement to be sent to the debug agent. Other targets involving entities within that group reply with group acknowledgements as soon as the breakpoint occurs. Acknowledgements provide a means of notifying the agent that (possibly) all entities within a breakpoint group are now halted. How a debug agent handles a group or global breakpoint is ultimately up to the design of the debug agent. The debug agent may use acknowledgements in any way appropriate to the design decisions made when a debug client is specified.

NOTE The occurrence of a DELAYED break event causes a GROUP-ACK message to be returned to the debug agent. The break identity returned to the agent is treated in the same manner as a group break event.

```
types
    Break-Identity ::
                kind : Breakpoint-Kind
                id : token ;

    Break-Entry ::
              ident : Break-Identity
              where : Address
              count : N
              pred-data : [Address]
              entity-group : Entity-set
              machine-group : Machine-Address-set;
```

The notion of a *group identifier* is encapsulated within the definition of a *Break-Identity*. A *Break-Identity* is used to name a breakpoint, this can be used to group entities *across* processor domains in such a manner that the breakpoint can be specified as affecting only that group of entities, the union of all entities over which a breakpoint identity is common. This is the definition of the *group breakpoint.*

The *count* field is used in conjunction with the breakpoint predicate mechanism. Predicates are truth-valued functions and as such cannot modify state. A typical predicate for an entity might assert that a value exists at a particular address after a procedure has been invoked. Used in conjunction with breakpoints and entities the predicate can perform watchpoint-like functionality. Because predicates are *Work-Handlers* they can be added and removed dynamically. When entities disappear the predicates associated with that entity namespace disappear. Setting and modifying breaks can be used to add or remove associated predicates that each entity has.

Entity predicates can be of arbitrary complexity depending on the assertion being made. A predicate takes *count* and *Address* arguments. The default predicate yields a counting breakpoint. If the count is zero the default predicate returns true. Also required for delayed break event handling is the "true" predicate—this ignores the state of count or predicate data and always returns true.

```
operations
Default-Predicate(args : Argument, ent-ws : Workspace) res : Data × Handler-Status

True-Predicate(args : Argument, ent-ws : Workspace) res : Data × Handler-Status
```

The last two types associated with breakpoints are the *Code-Address to Break-Entry* table and *Break-Identity to Break-Entry* table. Note that both map to sets of *Break-Entry.*

```
types
    Break =
         Break-Identity -m-> Break-Entry-set
              inv brk ≜
                  (∀ id ∈ dom brk ·
                      ∀ entry-1 ∈ brk(id),entry-2 ∈ brk(id) ·
                         let mk-Break-Entry(bid-1,addr-1,_,_,ents-1,_) = entry-1,
                             mk-Break-Entry(bid-2,addr-2,_,_,ents-2,_) = entry-2 in
                         (
                            (bid-1 = id) ∧
                            (bid-2 = id) ∧
                            (addr-1 ≠ addr-2) ⇒
                            (∀ ident-1 ∈ ents-1,ident-2 ∈ ents-2 · ident-1 ≠ ident-2)
                         )
                  )
```

The above invariant states that many breakpoint entries can have the same break identity as long as their addresses are not the same (then they would be the same breakpoint) and the entity sets are disjoint from each other. The breakpoint can be set to occur at an arbitrary number of addresses but the entities associated with that break entry cannot be used again for the same break identity.

```
types
  Code-Map =
          Address -m-> Break-Entry-set
            inv brk ≜
              (∀ addr ∈ dom brk ·
                ∀ entry-1 ∈ brk(addr),entry-2 ∈ brk(addr) ·
                  let mk-Break-Entry(bid-1,addr-1,_,_,ents-1,_) = entry-1,
                      mk-Break-Entry(bid-2,addr-2,_,_,ents-2,_) = entry-2 in
                  (
                    (addr-1 = addr) ∧
                    (addr-2 = addr) ∧
                    (bid-1 ≠ bid-2) ⇒
                    (∀ ident-1 ∈ ents-1,ident-2 ∈ ents-2 · ident-1 ≠ ident-2)
                  )
              )
```

The invariant for *Code-Map* states that an arbitrary number of breakpoint entries can be associated with one address so long as the entity sets of each breakpoint entry are disjoint. This is used to determine which breakpoint occurred at a given address with a particular entity previously active when a break event occurs.

## When Break Events Occur

Break events can be set to behave in three ways; they can affect the target machine for which they were set, a group of target machines or every target machine. The nature of the *Break-Reason* (defined in Chapter 6) determines the behaviour of the breakpoint handler. It is possible for a break event occurring locally to have a potential group or global effect if the breakpoint handler determines that the event was a group or global breakpoint. It is also possible that a monitor *receives* an external breakpoint message from another monitor or the debug agent which it *must* handle.

Each break event has a *predicate* associated with it. The predicate may be used to make assertions regarding the state of the entity. The predicate for a given breakpoint identity is entity *relative*. This means that the predicate executed as part of break event handling depends on the entity that was active at the time the event occurred. Each predicate for that entity is part of the namespace for that entity. The name of each predicate is the name of the breakpoint identity to which the entity is associated.

If the PUT is active when a break occurs, then the PUT *may* halt. If a global break is detected, any finalisation code associated with entities is executed before the PUT terminates. In effect this yields a *safe breakpoint*, a feature useful for debugging *safety-critical systems*. For example, an entity used for controlling a cooling valve to a nuclear reactor should be left open to maintain a stable state!

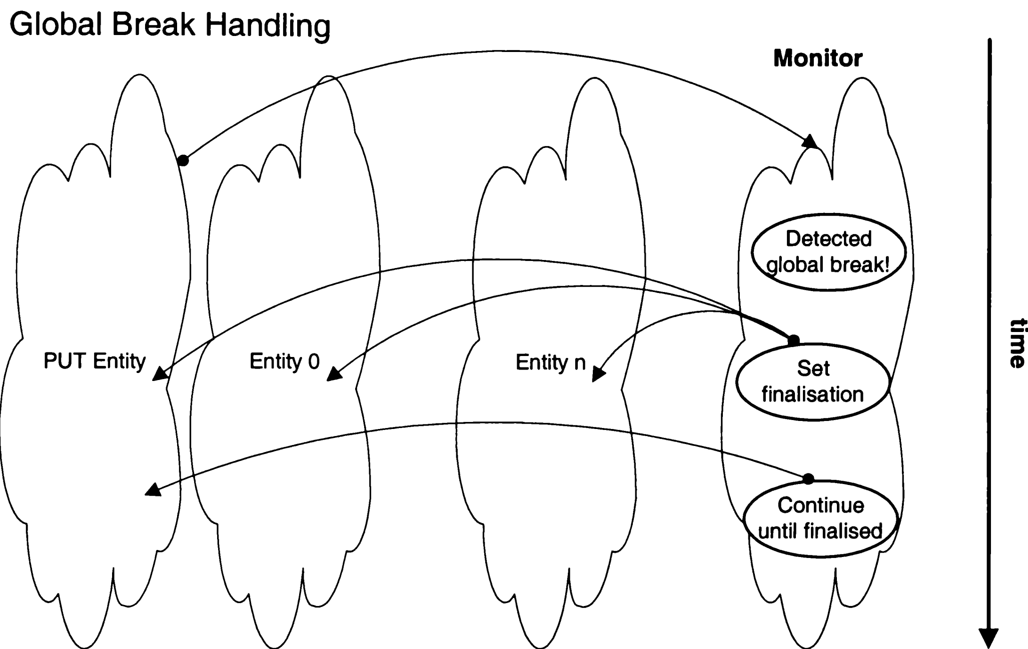



Fig. 5.14 *Global break event handling*

If a group breakpoint has been detected then the local target PUT entities are stopped. The group breakpoint message sent to other targets involved in the entity group acts as an indication that any active entities in the group are to be stopped.

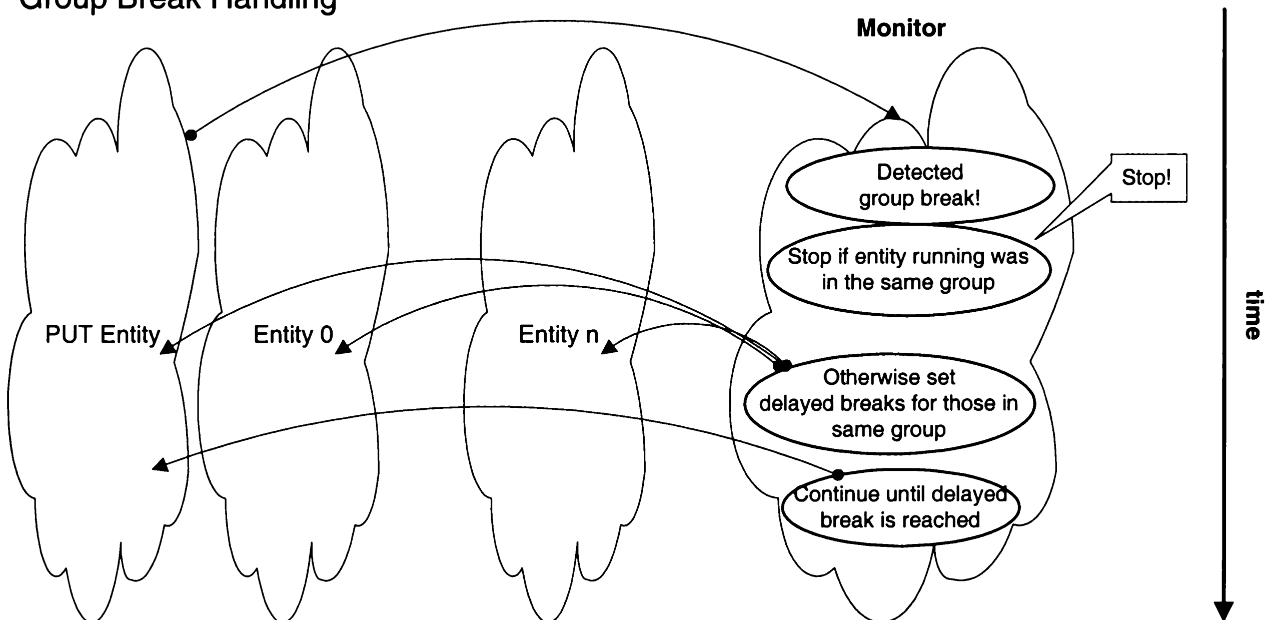



Fig. 5.15 *Group break event handling*

When a group break message arrives, activity on the receiving target is suspended (to handle the interrupt associated with the incoming message), as described in Chapter 6:—

a. If the PUT was active before suspension then—

(1) If an entity in the group associated with that group break identity was active then the group break is satisfied and the PUT is halted.

(2) If an entity that is not a member of the group was active then the group break location for the entities in the group is set to be the next instruction for each and all associated

entities are given the default predicate handler.

b. If the PUT was inactive, no further action is necessary.

NOTE The receipt of a group break message and subsequent detection of the group break do not generate further group break messages.

When a local break event occurs and a confirmed break event is detected, the reason for the occurrence could be one of three—a local break, a group break or a global break. The following steps take place:—

(1) The local break event handler determines *where* the break event occurred.

(2) Use the address to look up the *Code-Map* which yields a set of *Break-Entries* whose break events are associated with that address.

(3) Determine whether the entity which was running before the event is a member of the entity sets of any of the break entries. To preserve the data invariant on the *Code-Map* there must be at most one entity match or none at all.

a. If no match occurred then the breakpoint was *not* set for this entity and the PUT should be continued.

b. If there was a match the count is decremented by one and the predicate for that entity and breakpoint is executed.

i. If the predicate yields false then the PUT is continued.

ii. If the predicate yields true then use the kind field of the breakpoint identity field to determine the actual breakpoint kind.

c. A *LOCAL* break kind returns the breakpoint identity, local breakpoint acknowledge, address and entity information to the debug agent.

d. If it was a *GROUP* break kind and the machine set is empty (this is the result of a receiving group break message) then reply to the agent with breakpoint identity, group breakpoint acknowledge, address and entity information to the agent. If the machine set is non-empty then send group breakpoint messages to each machine in the machine set and reply with the breakpoint identity, group breakpoint acknowledge, address and entity information to the agent.

e. If the break kind is *GLOBAL* then reply with breakpoint identity, global breakpoint acknowledge, address and entity information to the agent then invoke the finalisation code (if any) for each entity.

f. If the break kind is *DELAYED* then the *True-Predicate* on that break identity for that entity is removed. If the break identity was "pushed" into the *saved-id* state of that entity then "rename" the break identity for that entry and set the *saved-id* state to nil. Otherwise remove the delayed break entry. In either case reply with the group breakpoint acknowledge, break identity (the delayed break identity is equivalent to a group break identity), address and entity information to the debug agent.

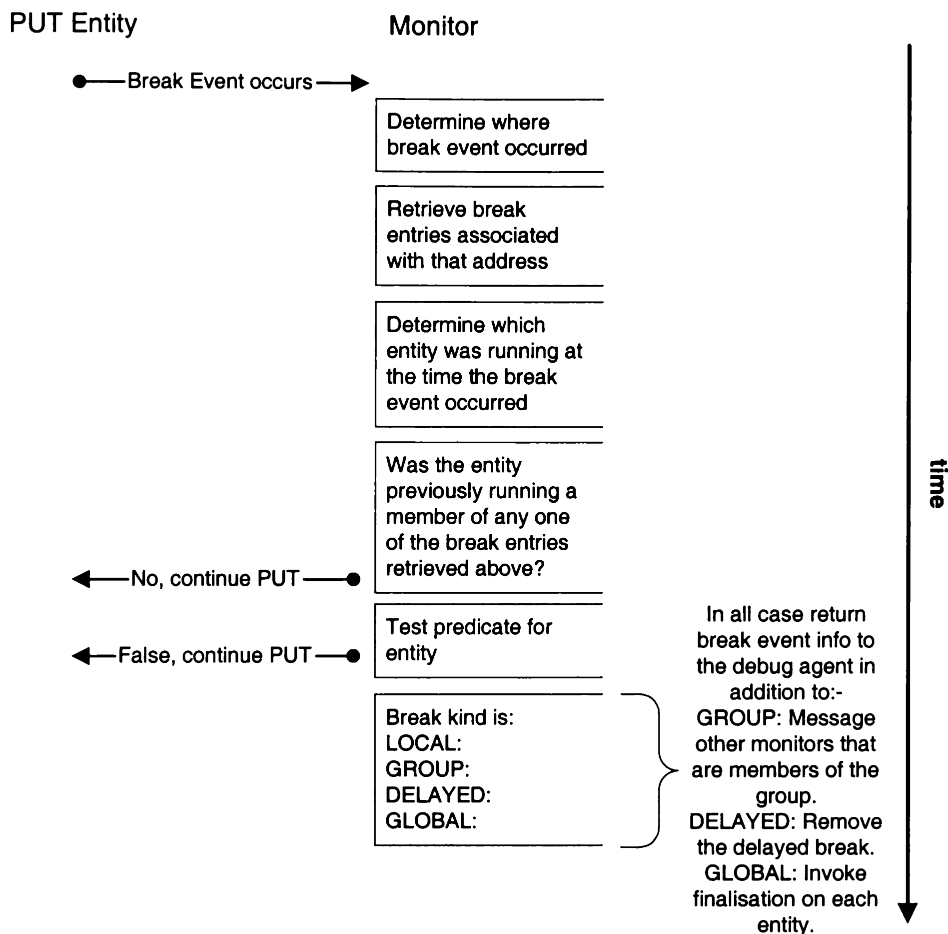



Fig. 5.16 *Local break event handling*

The *acknowledgement* used for reporting break occurrences provides an upper bound on the number of messages sent, particularly when a group breakpoint occurs across machines. In such a case, if no care were taken in setting up "delayed" breakpoints there would be a chance of *livelock* occurring. This means that potentially, the stable group break state for a given breakpoint identity might *never* occur. This is the reason for setting delayed group breaks—their occurrence implies that a group break of the same *name* previously happened. Provided this is done, a group break acknowledge will not cause further group break messages to be sent to members of the same group. If the debug agent requires a "satisfies group breakpoint relation" it could be implemented by waiting for *all* breakpoints in that machine set returning a group break acknowledge message for that *Break-Identity*. It is the debug agent's responsibility to maintain some information regarding "who" is part of the group breakpoint.

For the formal specifications of the break event handlers refer to Annex B.

### Setting and Clearing Break Events

A breakpoint is established using the *Set-Break* Work-Handler. The agent provides the address of the breakpoint, the identifier, kind, count and entity. No machine addresses are supplied to *Set-Break*, the *Modify-Break* Work-Handler is used to establish a group of machine addresses when specifying a group breakpoint across machines. Refer to Annex B for the detailed specifications of the *Set-Break*, *Modify-Break* and *Clear-Break* operations.

The *Set-Break* handler establishes a *Break-Entry* at a given address for a particular entity. The default predicate is added to the entity namespace supplied.

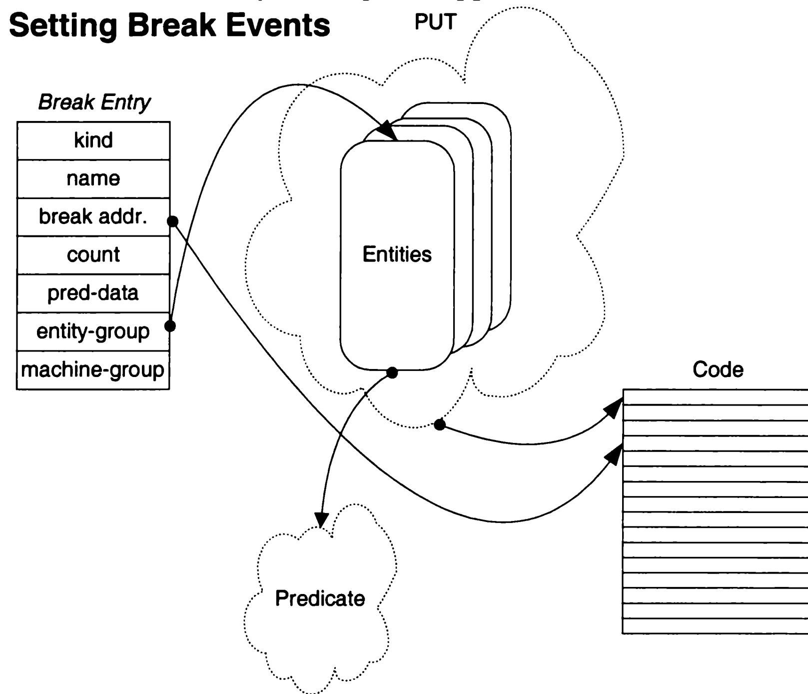



Fig. 5.17 *Setting Break Events*

The *Modify-Break* handler modifies an already existing *Break-Entry* adding a new entity to the break event or a machine address if a group breakpoint is being modified. The default predicate is replaced for the newly added entity.

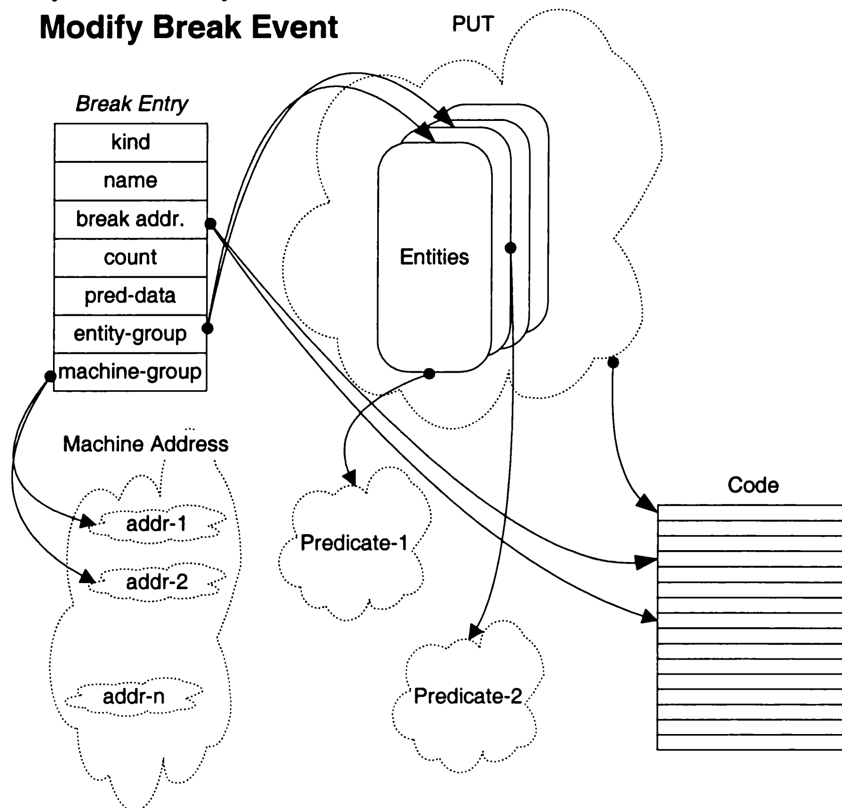



Fig. 5.18 *Modifying Break Events*

The *Clear-Break* handler finds a *Break-Entry* with the supplied address and removes all predicates from entity namespaces with that *Break-Identity.*

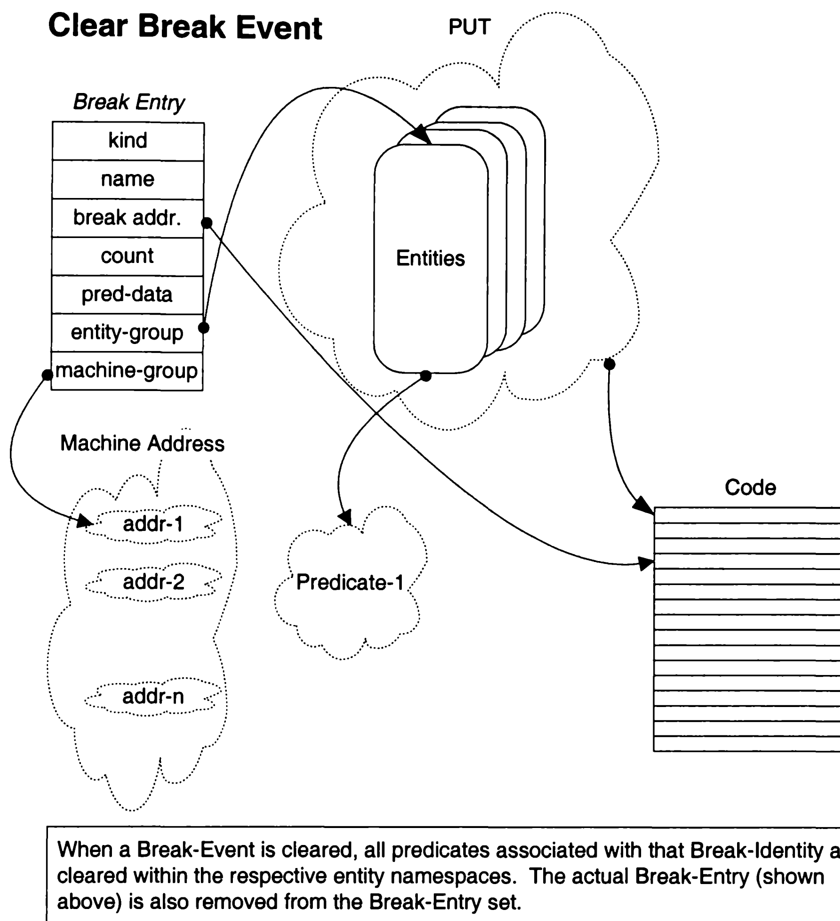



Fig. 5.19 *Clearing Break Events*

The method used to establish a break event is *target-dependent.* For example, special instructions may be inserted or specific registers loaded that cause a break event. Every time a break entry is set at an address for which there has been no previous break event set, target-dependent operations must be performed to establish that break event. Likewise in the situation where there are no entries set at a given address, the breakpoint should be cleared so that no further break event can occur there. In an implementation it is possible that the *Break-Handler* also may have to perform extra target-dependent operations depending on how the break events affect the instructions processed by the target CPU.

## Break Message Analysis

The occurrence of group and global breaks raises issues regarding message traffic and the maximum upper bound on messages that need to be sent to co-ordinate a group or global breakpoint. Reliable channels and "listening" monitors are assumed in this analysis. If a monitor does not receive a breakpoint message from another monitor then no retry is attempted by the sender. The sending monitor does not have a time-out/re-transmit mechanism for reasons described in Chapter 3. If the monitor that should be receiving a break message is listening on the wrong channel, that message is also lost. The analysis also assumes that the determination of a breakpoint and any necessary setup required to send messages to other monitors is negligible compared to network delays experienced by the message in transit.

Local break events which are determined to be of the local break kind are the easiest to analyse. The occurrence of such a break sends *one* reply back to the debug agent and returns control to the monitor. A local break is a *one phase* algorithm. The breakpoint is immediately satisfied and the

debug agent receives a message indicating that this break event has occurred and a breakpoint has been satisfied. If a local break occurs, only *one* message is sent to the agent.

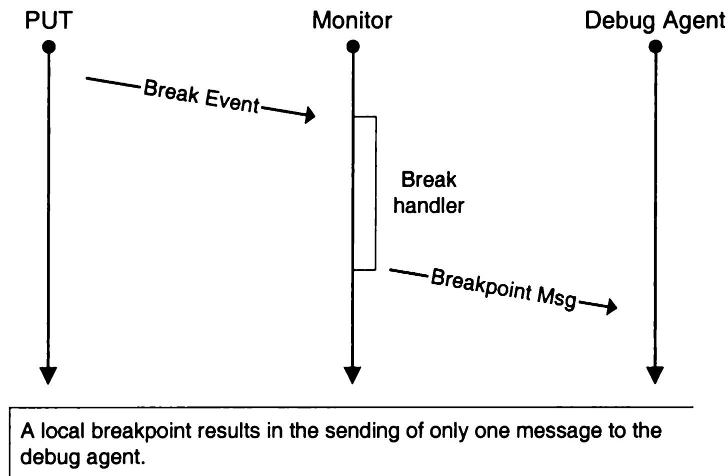


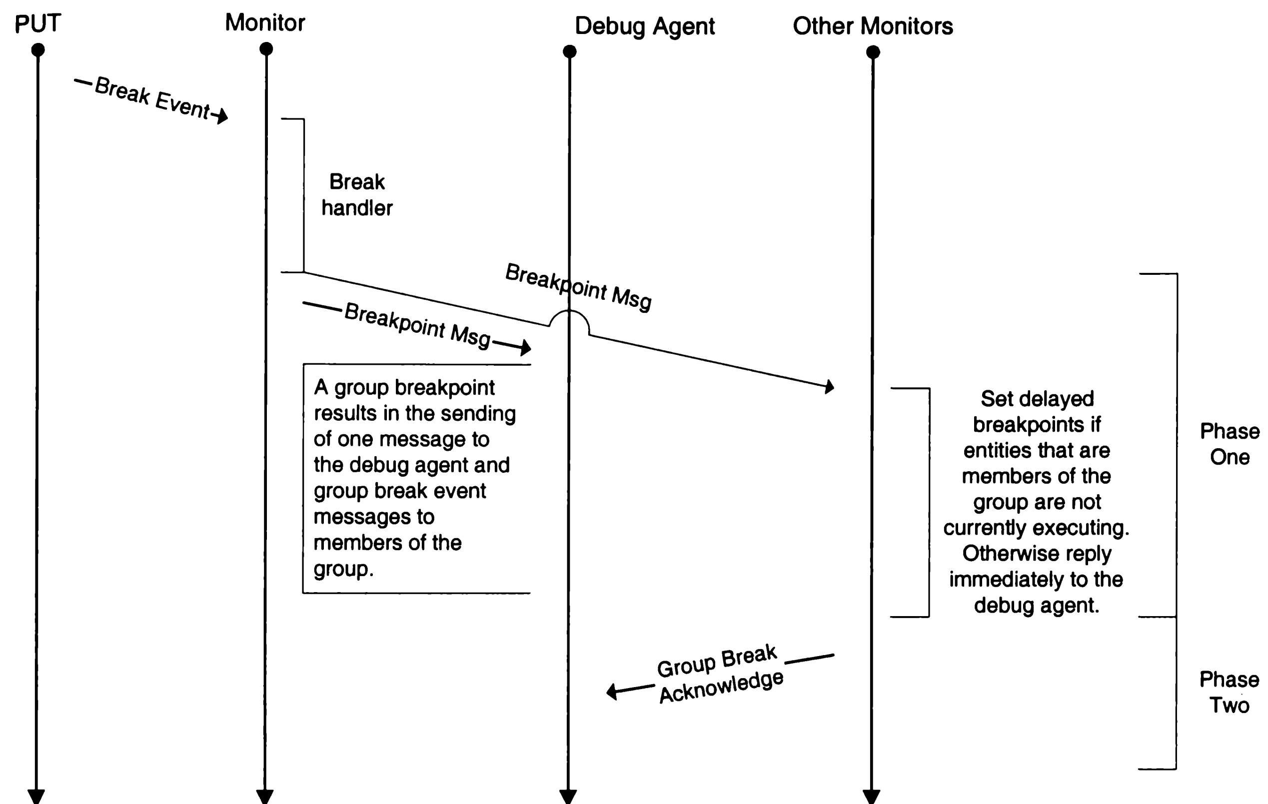

Fig. 5.20 *Local break event message analysis*

Probably the most difficult to analyse is the group breakpoint. The group break is divided into two phases. The first phase is the break event which occurs locally to a particular monitor and is determined to be a group breakpoint. A group break message is sent to each target in the set of machines participating in the group break while the initiator, having already satisfied the group break, replies to the debug agent. The second phase is while group break messages are being received by other members of the group. What happens is based upon whether the PUT is already stopped or the PUT was running and the entity previously executing was a member of the group or lastly that the PUT was running and the entity previously active was not part of the group. The first case results in no action being taken by the monitor—the PUT is already stopped. The second case requires an *immediate* response that the group break has occurred. This is the group break acknowledge to the agent. The third case requires that "delayed" group breaks be set for each entity in that group break. When these occur, the second phase of the group break is satisfied through a group break acknowledge message being sent to the debug agent. This arrangement implies the possibility that the second phase may *never* complete because those consequential breaks may *never* occur. This does not have any effect on the monitors concerned but may cause problems for the debug agent. Such occurrences may allow inferences to be made about the run-time behaviour of the PUT that could assist in its debugging.

> *If the debug agent is constructed in such a manner that the "satisfies group break" occurs* **only** *when all targets associated with that group stop then that break may never be satisfied.*

Best case message complexity occurs when *only one* group break of a given identity occurs locally. This means for $n$ targets at most there will be $2n - 1$ messages generated. The first phase will generate a reply to the agent by the monitor at which the group break occurred locally plus the remaining group break event indication messages sent to the $n - 1$ other targets in the group. The second phase (assuming all targets reach their group breakpoints) has $n - 1$ messages replied to the agent.

Group Break Event Message Complexity




The best case is where all members of the group breakpoint fire simultaneously. In this case it the maximum messages produced is 2n - 1. The worst case is when all members of group break point fire on the same breakpoint. This results in $n^2$ messages being produced.

Fig. 5.21 *Group break event message analysis*

The worst case scenario is a situation where **all** targets in that group have a group break occur locally simultaneously. The first phase then has each target attempting to send a reply to the agent and messages to each member of the target group. This means that each target sends $n$ messages for a worst case of $n^2$ messages being sent for a group. Note that from the debug agent's point of view this effectively satisfies the group breakpoint immediately if the debug agent is implemented in such a manner that requires all members of the group to stop before the group breakpoint is satisfied. The second phase becomes redundant because each target monitor will have *stopped* during their first phase and therefore ignore any incoming group break messages.

The global breakpoint uses the *debug agent* to do any message forwarding. Its analysis is identical to the best case for group breakpoints, $2n - 1$. There is one reply from the *initiator* of the global break, $n - 1$ messages sent from the debug agent to all other monitors and $n - 1$ messages in reply to the original global break notification message sent by the debug agent. While replies will be sent in response to the global break event, the PUT only halts when all entities have completed their finalisation and the PUT actually terminates. For reasons given previously, this termination may never occur if there are problems with finalisation. In this situation, the debug agent may *explicitly* send another global break notification to the target that is executing finalisation code—this behaviour is then equivalent to the debug agent *stopping* the PUT. The results of the termination phase are separate from the initial reply to the global break and are not included as part of the calculation given in this paragraph.

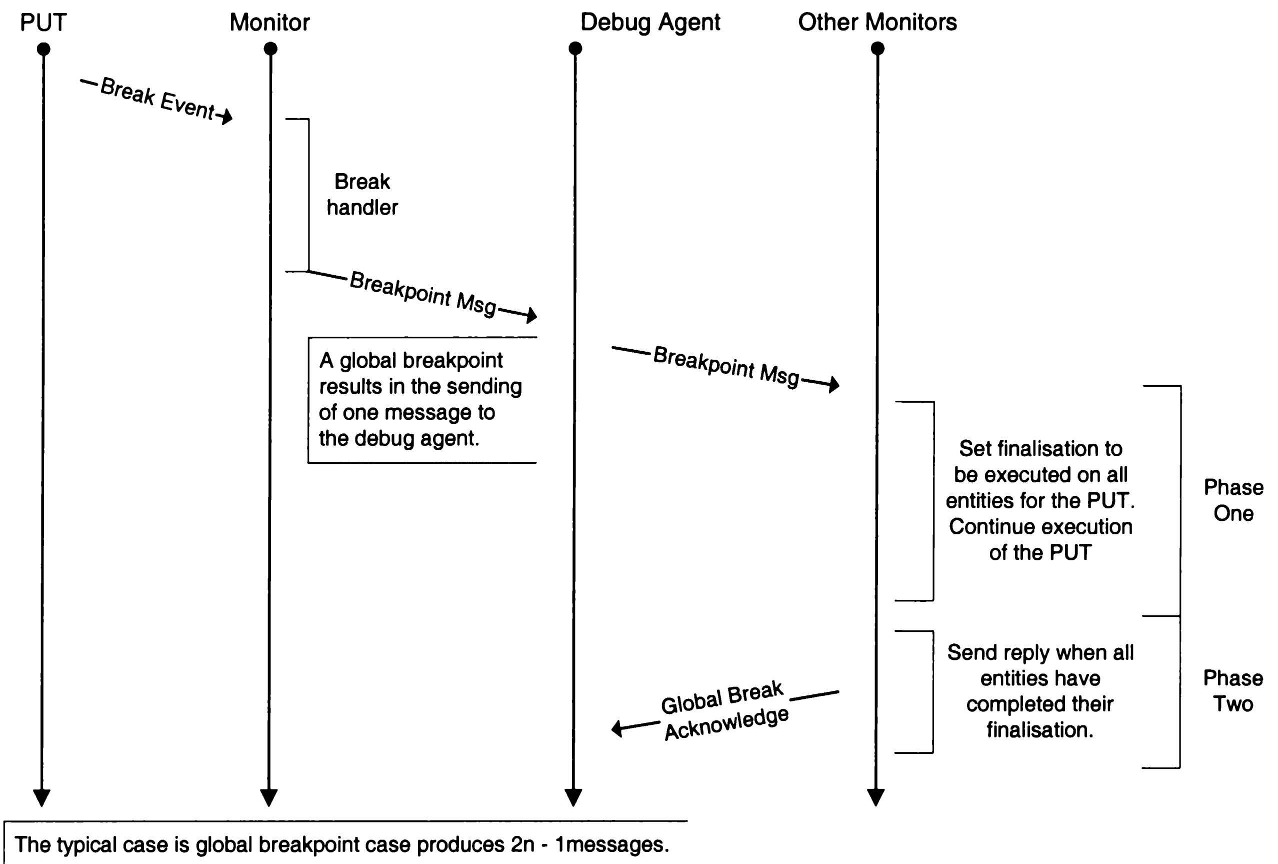



Fig. 5.22 *Global break event message analysis*

The real difference between the analysis of the group and global break message complexity lies in the overall effect of the break. A group break can affect only a cluster of targets. For a group break in which the total number of targets in the distributed system, $m$, a group of targets in that same system, $n$ where $n \leq m$ in the best case will have less messages sent than the global case where **all** targets are affected because they are all subject to message traffic.

## The Good and the Not So Good

The advantages of the namespace model have to be weighed against a few complications that are not readily apparent from the description given in this chapter. The complications are not extreme enough to be described as disadvantages but they do require further explanation:—

a. The configuration phase is *very* important. Session establishment not only initiates a dialogue between debugger and monitor but also obtains any resource configuration information on the target system so that the debugger can find the appropriate configuration descriptions. It is critical that the debug agent use this configuration information to allow the namespace model to be used to its greatest advantage by both the debug agent and monitor server.

b. As device descriptors from dynamic configuration are sent to the debugger, symbol descriptions representing that resource are sought by the debug agent assuming they *exist*. What valid assumptions can be made in this case, about the resource involved? If they do not exist the debug agent has very little information that it can use to access that namespace. This means that debug agent interface to a namespace is extremely important—it must exist before any useful debugging can be achieved.

c. On an implementation related note, symbol descriptors would have to be provided by the vendors of the hardware for that target machine. Therefore, this debugger protocol, at least from the user's point of view requires the co-operation of vendors producing symbol descriptors describing the structures of the various namespaces. Where special handler semantics are required, then implementations for a range of CPUs is required. The explicit description and derivation of these structures is not part of this research.

d. Since handlers can be added and removed dynamically, the workspace needed by the monitor requires more complex memory management. This is of course ultimately an implementation issue. Hardware with a limited address space could be restricted by such requirements.

e. The resource configuration described *ideally* requires hardware support. The target hardware on which this monitor resides would be able to auto-configure at cold/warm start-up as well as handle insertion (and auto-configuration) of hardware modules (expansion cards etc) at run-time.

The main advantages in using the namespace to model resources and entities in the monitoring system are:—

a. Security, the structure of a namespace is duplicated exactly by the matching symbol descriptor that describes the form of that namespace. Invalid requests to namespaces are rare because checking can be done by the debugger through the symbol template that is part of the description of target resources.

b. Dynamic hardware configuration is simpler. Namespaces represent hardware objects. When hardware dynamically configures, the monitor sees this as the addition of a namespace. In this way, debug agent and monitor server are always "synchronised" from a resource point of view. Conversely, if the hardware supports it, devices can be removed and their corresponding namespaces will also be removed.

c. The principles behind the namespace concept are simple and highly regular, not only can hardware be easily modelled but also monitor system software structures. While the namespace abstraction is very simple, because the interpretation of packet requests from the debug agent is left up to the namespace handlers it is extremely flexible.

d. The model handles the notions of heterogeneous machine domains well. Symbol descriptors provide mappings to the relevant namespaces which cause activations of the namespace handlers at the monitor. The handler is all important; whatever it does is dependent on the namespace which it represents.

e. It is useful to note that *all* requests to the monitor are namespace requests. Session establishment, control of program execution and the setting of breakpoints among others things are accomplished through requests to namespaces. This uniformity considerably simplifies protocol design.

f. The construction of namespace handlers encourages their potential code *reuse* in a multi-configuration environment. This is especially true of handlers that get downloaded from the debug agent to the monitor server when dynamic configuration occurs.

g. A novel concept that "comes for free" because of the namespace model is that of dynamic modification of handlers. In the case of breakpoint handlers this can be used to tailor breakpoint handler algorithms for particular situations. If the debugger user does not like the method in which a service is handled, they can always write their own and replace the previous handler! The only limitation is that the user defined handler must obey the previous calling interface

(arguments) and the Work-Handler signature depending on what operation the user defined handler is performing. This limitation applies only to handlers that are invoked by event handlers such as trap and interrupt co-routines because the debug agent can *never* invoke the break handler explicitly through a message being sent. If a handler is always invoked by the debug agent then that interface can change because the debug agent also knows about the calling interface change due to a replacement taking place.

h. Somewhat stranger still, the possibility exists for the monitor server to debug itself! User defined handlers can be installed into a particular namespace and executed. If the result of the user defined handler is a trap, rudimentary debugging information exists that allows determination of where the error occurred so that debugging of handlers can take place in addition to the debugging of user programs.

# Chapter 6

# Driving the Model

The monitor consists of four major interacting components that are modelled as *co-routines.* The co-routines provide a framework that:—

a. React and respond to both internal and external events that are useful to the monitor.

b. Provide support for the namespace model. This support takes the form of an execution environment in which the namespace model can be hosted to acquire and process messages and events.

The co-routines and event processing operations described are parts of the *functional core adapter* as opposed to the *functional core* (see Figure 1.1).

The detailed definitions of the co-routines and auxiliary functions discussed in this chapter are presented in Annex C.

### Co-routines and Continuations

The operation of a co-routine relies on the ability to be *transferred from* and *resumed by* other co-routines. A transfer to another co-routine occurs in two ways; a voluntary synchronous yield where the transfer takes place under the control of the yielder or an involuntary asynchronous transfer from a yielder to an *interrupt handler.*

The contents of this section are based upon the the continuations and co-routines model presented in the Modula-2 Standard [ISO94] document. The definitions presented here, however, simplify some parts of that model. For example, Modula-2 allows continuation of program activity when an *exception* occurs. This is not required functionality for a monitor! In the case where such exceptions occur during the execution of the PUT, it is up to the human user to decide whether the PUT should remain halted or be allowed to continue.

The Modula-2 Standard formally specifies the semantics of *continuations* which are used extensively by the run-time Modula-2 machine to support the COROUTINES library which is part of the standard definition. For further information regarding the semantics of continuations and co-routines, Milne and Strachey [Mil76] provides a detailed reference.

**types**

$$Coroutine\text{-}storage = Coroutine \xrightarrow{m} Coroutine\text{-}store;$$

$$Coroutine\text{-}store :: store : Store$$

The *Coroutine-Identity* and *Loc* can be defined as:—

types

*Coroutine* =
⟨*IDLER*⟩ |
⟨*TRAP*⟩ |
⟨*INTERRUPT*⟩ |
⟨*MONITOR*⟩ |
⟨*PROGRAM-UNDER-TEST*⟩;

*Loc* =
token;

All co-routines require workspace in which to function. This workspace provides any necessary area to provide for saving context, maintaining local variables and procedure activations. All co-routines have the following attributes:—

a. Workspace base and size denoting the area in which context is saved and a per co-routine stack for maintaining call frames and local state.

b. An initial *program counter* denoting the position of the first instruction at which execution is to commence.

c. An initial *workspace pointer* denoting the start of the stack area. The area referred to by this object is necessarily bounded within the workspace area described above.

d. A register save area within the workspace but exclusive of the stack area used to save and restore execution state when transfers of control take place between co-routines.

Each of the co-routines named above requires these values to be established at the time when initialisation of the monitor takes place. An exception to the rule is the PROGRAM-UNDER-TEST which is established by the *Initialise-Program* handler of Target namespace.

Co-routines to handle events are handlers attached to sources of interrupts. The actual mapping of an *Interrupt-source* is necessarily target defined. Some microprocessors such as the Motorola 680xx family of CPUs use a *vector table*. This allows a handler to be associated with a particular source of interrupt, a one-to-one mapping where each interrupt handler performs the task which the hardware device associates with that vector. Other microprocessors, notably those designed around a RISC implementation might have only one source of interrupts. In this case it is up to the interrupt handler to determine the *cause* of the interrupt and have the handler's operation determined by the cause of the interrupt.

types

*Handlers* =
$\mathit{Interrupt\text{-}source} \xrightarrow{m} \mathit{Handler}$;

*Interrupt-source* =
token;

*Handler* =
*Coroutine*

Monitor initialisation routines require a means of attaching event handlers to the sources of those events. An *Interrupt-source* provides a handler the ability to dispatch depending on the *reason* for the asynchronous event.

The monitor is a set of co-routines, two of the co-routines are *attached* to event sources; interrupts and traps:—

a. Traps—a *synchronous* internally produced event that causes a change of control flow from the context originally executing at the time the trap occurred. Traps typically arise from erroneous conditions in the PUT such as division by zero and exceeding address bounds.

b. Interrupts—an *asynchronous* externally produced event that also causes a change of control flow from the context originally executing at the time the interrupt occurred. Interrupts typically occur due to changes in state external to the processor.

While traps and interrupts may be sourced differently, both have the same effect in that they change the flow of execution to handle the respective event.

The PUT may also handle its own traps. Traps not handled by the PUT are handled by the default trap handler which causes a transfer of control back to the monitor. If the PUT replaces handlers for various traps, it is the responsibility of the monitor to restore any trap handlers previously installed for use by the monitor. Conversely, the monitor should also restore the handlers provided by the PUT when the debugger continues the PUT. Desirably, each co-routine should have its own trap table with the trap table of the PUT being a copy of the monitor's. The PUT can then replace the handlers of various sources with its own.

operations
*attach : Interrupt-source* × *Coroutine* $\xrightarrow{o}$ *()*
*detach : Interrupt-source* × *Coroutine* $\xrightarrow{o}$ *()*

The operations *attach* and *detach* allow interrupt driven communication devices to be changed during a session to accommodate situations where the PUT *may* use the communications device that the monitor was using for communication with the agent. If another communications device is available for use, the monitor will use this channel so that messaging can still take place even if the PUT is using the channel previously used by the monitor.

operations
*interrupt-occurs : Interrupt-source* $\xrightarrow{o}$ *()*
*transfer-to-handler : Interrupt-source* $\xrightarrow{o}$ *Coroutine*

When asynchronous events (interrupts) or synchronous events (traps) occur the operations *interrupt-occurs* and *transfer-to-handler* are defined to provide a means of non-preemptively switching between co-routine activations to which those co-routines have previously been attached.

operations
*transfer-to-coroutine : Coroutine* $\xrightarrow{o}$ *Coroutine*

When co-routines are required to transfer control between themselves a *transfer-to-coroutine* is required. Note that unlike *transfer-to-handler*, co-routine transfer is synchronous.

*Co-routines* and *continuations* provide an execution environment for the PUT as a co-routine in addition to those which compose the monitor software. Provided a co-routine's state is preserved, an asynchronous or synchronous transfer of control may take place with control being eventually returned to the transferrer at the point where the transfer occurred. The return of control is transparent as far as the co-routine is concerned and as such represents a continuation of control flow even though computation was suspended during the transfer to the other co-routine.

types
*Coroutine-env ::*
*mcont : Coroutine* $\xrightarrow{m}$ *Program-cont*
*caller : Coroutine;*

A *Program-cont* consists of all component parts of a program which is a *Coroutine.*

**types**

$$Program\text{-}cont = Cmdcont \rightarrow Coroutine\text{-}env \rightarrow Statecont\ ;$$

$$Cmdcont = Coroutine\text{-}env \rightarrow Statecont\ ;$$

$$Statecont = State \rightarrow Answer\ ;$$

$$State = Context;$$

$$Answer = External\text{-}state\ ;$$

The following operation definitions provide functionality similar to that provided by the Modula-2 COROUTINES library module. The operation signatures defined are:—

**operations**

$initialise : Coroutine \times Program\text{-}cont \xrightarrow{o} Coroutine\text{-}env$
$resume : Coroutine \xrightarrow{o} Coroutine$
$current\text{-}coroutine : () \xrightarrow{o} Coroutine$

(1) *initialise*—creates a new co-routine.

(2) *resume*—is used to explicitly *transfer* from one co-routine to another synchronously.

(3) *current-coroutine*—identifies the currently active co-routine.

These primitives are all that is required to initialise the monitor co-routines and also transfer control either by the explicit use of *resume* or through an implicit *resume* that takes place via a *transfer-to-handler* when an asynchronous transfer of control takes place due to interrupts or traps occurring.

## The Co-routine Stack

The reasoning behind the use of a co-routine stack is to do with the handling of *external* break events. To explain the reason why it is needed, an example of the problems caused if there were no such mechanism in place will be given. Consider the following break event handling situation:—

> A local break event occurs. This causes some trap processing to take place that eventually causes a breakpoint handling message to be constructed and *pushed* to the *head* of the message queue. The trap co-routine then transfers control to the monitor co-routine so that break event processing can occur. The break handler is immediately invoked (through the namespace finder mechanism) to process the break event. The break event handler can *either* stop the PUT if the conditions governing a break event are satisfied *or* continue the PUT. The situation that arises is—"What happens when an external break message arrives during break event handling?". Traps *always* push messages to the head of the queue. If an external break event occurs while the break handler is handling the first break then the message pushed could be "ignored" if the message pushed by the trap co-routine occurred *before* the break handler's *continue* message was pushed to the queue.

This situation can only occur if the break event is determined *not* to have occurred, ie., when a predicate evaluates to *false*. If the external break message arrives *after* the continue message has been pushed to the head of the queue, then results are just as problematic. The possibility in this situation is that too many *continue* messages could be pushed onto the queue resulting in erroneous behaviour as each message is processed. For example, it is possible that one of the break events results in the PUT being halted, and one of the previous invocations of the break handler could have pushed a continue message meaning that an undesired fake continue message could be processed by the monitor.

In the situation where a break is determined to have occurred the above does not apply because the STOPPED state has been established by the time a later occurring trap message has been pushed. Since no messages are pushed by the monitor when a break event is determined to have occurred there are no message queue serialisation problems. A stopped PUT cannot be stopped any further!

The break event handler of Target namespace uses the co-routine stack to determine the co-routine last active. If the last active co-routine was the Monitor co-routine itself then messages to continue the PUT are *not* enqueued. If external break messages keep arriving while the monitor is active the co-routine stack will ensure that there will only ever be *one* legitimate continue message pushed onto the queue irrespective of whether one or more break events occur.

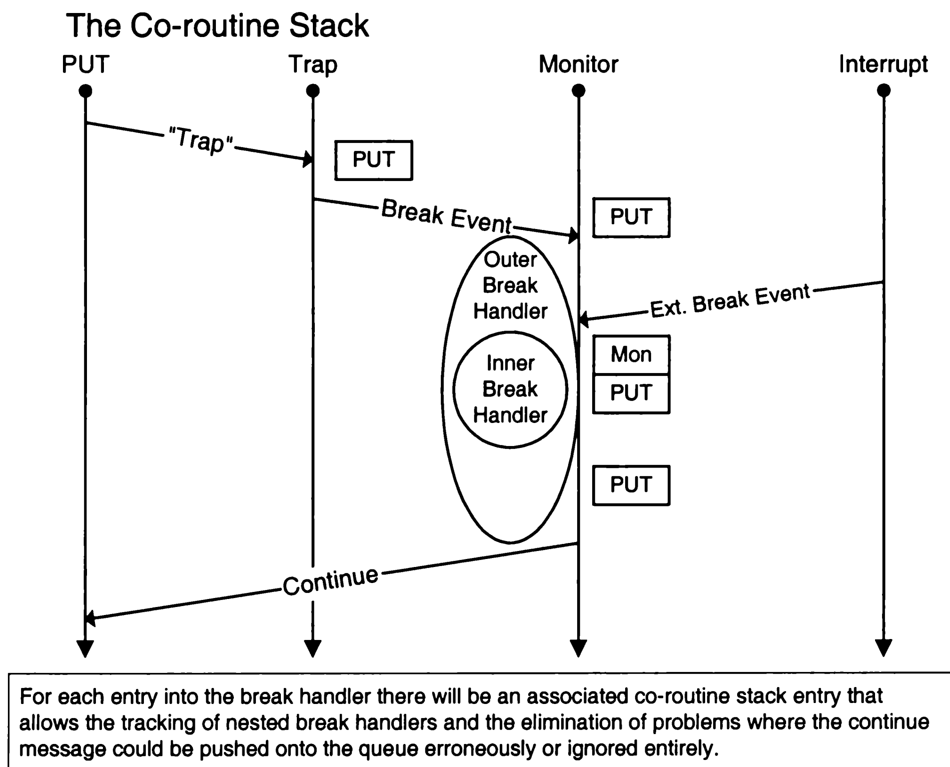



Fig. 6.1 *The Co-routine Stack—An Application*

The types and operations defined upon the co-routine stack which is a part of monitor global state are:—

types

$CR\text{-}Stack =$

$\quad Coroutine^*;$

operations

$Push\text{-}CRStack(cr : Coroutine)$

$Pop\text{-}CRStack()\ cr : Coroutine$

### Storage Management

The namespace model and monitor framework require several auxiliary functions to provide memory management primitives—used to manage heap and allow workspaces and namespace environments to be created and destroyed.

The memory management operations specified in this section are based upon the abstract storage model defined by the Modula-2 Language International Standard [ISO94].

types
*Store* ::
*pool* : *Storage-pool*
*vals* : *Workspace-store*
inv *mk-Store(pool,values)* $\triangleq$
*pool-locs(pool)* $\bigcap$ dom *values* = {};

The invariant requires that the storage locations allocated to a co-routine are disjoint from the storage locations used by other co-routines. *Workspace-store* is separate "state" from co-routine store, it can be viewed as space set aside for the construction of namespace environments. In terms of the monitor execution environment, there is storage provided for co-routine state such as the stack of each calling co-routine and the co-routine closure.

types
*Storage-pool* ::
*allocated* : *Address*
*rest* : token;

The *rest* component is an implementation-dependent type consisting of a set of *Loc*, possibly derived from the *allocated* component, from which storage can be allocated.

types
*Workspace-store* =
token $\xrightarrow{m}$ *Storable-value* ;
*Storable-value* =
token |
$\langle$*UNDEFINED*$\rangle$ ;

The next operations *generate-loc* and *get-locs* manipulate *Loc* storage objects within the *Store*. The *generate-loc* operation generates a new storage location (one currently not within the domain of the storage map) and sets its value to undefined.

operations
*generate-loc(store : Store) res :* token × *Store*
post
*1.* let *loc-token* : token be st *loc-token* $\notin$ *get-locs(st)* in
*2.* let *mk-Store(pool,values)* = *store* in
*3.* let *nvalues* = *values* $\uplus$ {*loc-token* $\mapsto$ UNDEFINED} in
*4.* *res* = *mk-(loc-token,mk-Store(pool,nvalues))*

The *get-locs* operation allocates a set of tokens from the available storage pool.

operations
*get-locs* : *Store* $\rightarrow$ token-set
*get-locs(store)* $\triangleq$
*1.* let *mk-Store(pool,values)* = *store* in
*2.* *pool-locs(pool)* − dom *values* ;

The *pool-locs* function returns an implementation-dependent set of *Locs* derived from the storage pool.

operations
*pool-locs* : *Storage-pool* → token-set
*pool-locs(pool)* ≜
(
  implementation-dependent
)

The *deallocate-locs* function returns the set of storage locations (currently in the domain of the storage map) to the storage pool.

functions
*deallocate-locs* : token-set × *Store* → *Store*
*deallocate-locs(locs,store)* ≜
1. (
2.     let *mk-Store(pool,vals)* = *store* in
3.         let *nvalues* = *locs* ⩤ *vals* in
4.             *mk-Store(pool,nvalues)*
5. )

Three operations are defined for use in the allocation and deallocation of workspaces as used by the namespace model. The *Workspace* represents the state area of the namespace environment but also the namespace map and the handlers for that namespace.

operations
*Acquire-Space* : $\mathbb{N}_1$ → *Workspace*
*Release-Space* : *Workspace* → **B**
*Fill* : *Workspace* × **N** × token$^+$ → *[Work-Handler]*

a. *Acquire-Space*—returns a reference to a *WorkSpace* of a specified size.

b. *Release-Space*—returns the given workspace to the free pool from which namespace environments are allocated.

c. *Fill*—takes a workspace, a sequence of tokens and copies the token values into the workspace.

The *Workspace* is an integral component of each namespace. It consists of a base address denoting the start of the namespace as it would exist in memory, the size of the workspace and the state area of that namespace used for maintaining the local state of the namespace object. This pair of base address, size and state area are used to manage the workspace so that handlers, map and state area are kept separate.

## Co-routine Structure

NOTE The definitions of these co-routines are given using the executable subset of *VDM-SL*, that is, each of these co-routines is given in explicit form rather than the implicit specifications presented so far.

The namespace model processes messages and returns results to the invoker of the handler, which in many cases is the debug agent. While the namespace model is a pivotal part of the debug protocol, software must exist at debug agent and monitor to provide an *environment* in which messages can be sent and replies received from the namespace object tree. The monitor must provide a *framework* to support the namespace model. It is the responsibility of the monitor to pass messages to the namespace engine for processing. These messages are acquired as the result of a debug agent's invocation of handlers and to event occurrences local to the target such as traps that may have resulted from the execution of the PUT.

The monitor can be logically divided into two kinds of components; one group that produces messages and another that consumes messages. The *Message-Queue* is used to connect message producer to message consumer. The monitor is typically a message consumer although handlers can also produce messages for other namespace handlers to consume. The monitor removes messages from the queue and processes these messages in *FIFO* order. The messages are then processed by the namespace engine.

No differentiation is made between the way in which the monitor handles messages and the way it handles events. An *event* refers to any locally (or externally) sourced message that has no corresponding request by the debug agent. An example of an event would be the occurrence of a trap such as a data alignment error or divide by zero. While the occurrence of an event has no corresponding debug agent request, the reply that results from that event being processed by the namespace engine will be sent back to the debug agent provided a connection exists. Monitors that exist on other target hardware in the system under test may also be sent messages—the group breakpoint handling described in the previous chapter is an example of such behaviour.

The message producers are the *Trap* and *Interrupt* co-routines, the consumer is the *Monitor* co-routine. In addition to these three there is an *Idler* co-routine whose use is required when initialising the monitor upon start-up as well as for managing message flow control. Collectively these four co-routines *are* the *functional core adaptor* of the monitor.

The co-routines which make up the framework in which monitor actions take place are:—

(1) *The Interrupt* co-routine processes messages that affect the monitor and, indirectly, the debug agent. Events such as messages to the Monitor co-routine are retrieved by device drivers responsible for retrieving messages from the debug agent and other monitors. Other events handled include the dynamic configuration of hardware devices which cause additions or removals of namespaces to take place. Interrupts typically occur in *external* events such as the arrival of a message packet over a network interface.

(2) *The Trap* co-routine catches all synchronous (in the context of the execution of the instruction stream currently executing) events that cannot be handled by the co-routine currently executing. These events consist of potentially fatal errors such as address errors, bus errors and others.

(3) *The Monitor* co-routine could be described as the *queue-handler* or *monitor namespace engine*. While the namespace tree does not belong to any particular co-routine (it is a part of global state), most handler invocations take place via the namespace dispatch mechanism that is part of the Monitor co-routine. This co-routine is a message consumer responsible for *sending* any replies to the debug agent or other monitors.

(4) *The Idler* co-routine which in essence does nothing! It is, however, important to the functioning of the monitor software as a whole because the Idler co-routine runs when nothing else can run. When the Monitor co-routine has no messages to process and/or no program-under-test is executing, the Idler co-routine runs. The Idler is also used to bootstrap the monitor software at cold start.

Finally it must also be remembered that the *Program-Under-Test (PUT)* is itself a co-routine! When the PUT is loaded an execution context is constructed—a co-routine. The monitor starts the PUT by *transferring control* to that co-routine. The PUT is free to implement any tasking model that is required by the embedded application for a particular situation. It must be stressed that the co-routines discussed in this section need not necessarily be of the same kind used to implement the PUT.

Figure 6.2 illustrates the relationship between all co-routines that are part of the monitor software. The arrows in the diagram illustrate transfers of control between co-routines. Traps and interrupts cause asynchronous transfers of control to take place. The activation of a program-under-test by the monitor and its eventual return are examples of synchronous transfer of control.

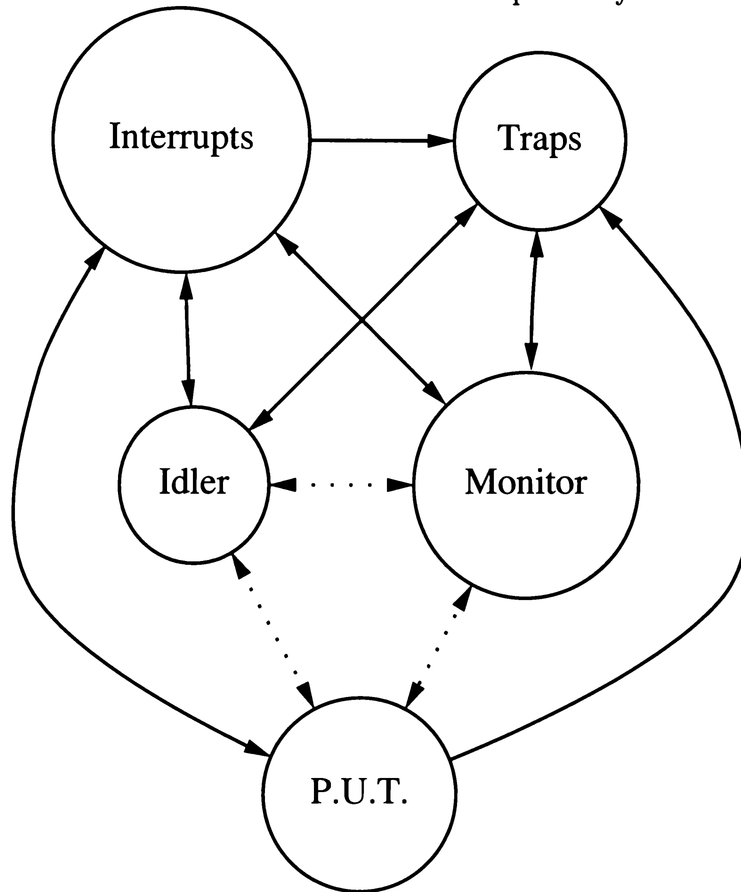



Fig. 6.2 *Transfers of control between monitor co-routines*

The dotted lines on the diagram represent *synchronous* transfers of control between co-routines, the operation that starts the PUT is a synchronous transfer of control from the Monitor co-routine to the PUT co-routine. The solid lines represent *asynchronous* transfers of control that might result from an interrupt or trap occurring. Ideally, the arrow from Monitor to Trap co-routine should not be required in a correct implementation of this model. However, since the monitor allows user defined handlers to be written for a given namespace and added to the namespace framework there is the possibility that such a handler is erroneous. It is user defined handlers which force the design to have to deal with traps caused by handler invocations running in the context of the monitor co-routine.

In the case where a user defined Work-Handler causes a trap, *normal* trap processing will occur. Instead of expecting a reply for the request made by the debug agent, the agent will instead receive a trap message indicating where the trap occurred and *who* caused it—the monitor is an entity just like the PUT. The debug agent receiving the trap message will be able to infer in such a case that the offending namespace handler requires some debugging!

## Monitor Co-routines and Global State

All co-routines have access to the *Root* namespace environment. Co-routines and the Root namespace are part of monitor global state. The "top" of the namespace hierarchy is $\rho_{root}$. As Root namespace exists at the top of the namespace hierarchy, it is the *only* namespace that has access to other namespace environments through its access to the map of the Root environment. Root

namespace could be considered to describe the *base* namespace environment with other subordinate namespaces being *specialisations* of Root namespace both in the state maintained and the handlers that act upon that state. All other environments subordinate to Root namespace are *indirectly* accessible via the environment component ($\rho_{root}$) of the monitor state. For a given namespace environment there *always* exists a path from the Root environment to a handler within a child environment in the namespace tree. This also implies that all co-routines have indirect access to all namespace environments beneath the root environment as all co-routines have access to monitor global state (and hence $\rho_{root}$).

Monitor global state contains all necessary state to maintain execution contexts for the monitor co-routines, the global message queue and the *Raw-Context* of the last interrupted co-routine. The global message queue is used by both Trap and Interrupt co-routines. The Monitor co-routine *always* processes messages *enqueued* by either the Trap or Interrupt co-routine, it never receives or handles them directly.

Global state can be subdivided into three categories; the Root environment, monitor co-routine descriptors/last interrupted context and the message queue used by event and message producers such as the trap co-routine for use by the monitor co-routine message consumer:—

a. The Root environment ($\rho_{root}$) is the top of the namespace hierarchy. Various namespaces such as External, Devices, Communication Devices and Target are environments subordinate to $\rho_{root}$. While each is its own separate environment, the monitor's state is composed of this namespace object tree.

b. Co-routine descriptors are also required so that all co-routines have access to the descriptors of each co-routine when a transfer is required, for example, between Trap co-routine and Monitor co-routine. The PUT co-routine is maintained as part of Target namespace. State is also required to maintain the reason why an event occurred as well as providing access to the last interrupted entity.

c. The global message queue is the channel through which all messaging from the debugger and other monitors takes place. Individual messages on the queue are processed in FIFO order by the Monitor co-routine with each message being a name path sequence to the handler in addition to the arguments provided to the handler.

Environments can be added or removed from the Root namespace but there are *no* Work-Handlers within Root namespace. Root namespace maintains no state information as a result of this. Because the Root map refers to the subordinate namespace environments below Root namespace any required initialisation of subordinate namespaces will also need to be done as part of initialisation.

state *Monitor* of
$\rho_{root}$ : *Env*
$\rho_{dev}$ : *Env*
$\rho_{memory}$ : *Env*
$\rho_{cpu}$ : *Env*
$\rho_{tgt}$ : *Env*
$\rho_{ext}$ : *Env*
$\rho_{driver}$ : *Env*
*monitor-context* : *Context*
*mon-ws-base* : *Address*
*mon-ws-size* : $\mathbf{N}_1$
*idler-context*: *Context*
*idl-ws-base* : *Address*
*idl-ws-size* : $\mathbf{N}_1$

```
    trap-context : Context
    trap-ws-base : Address
    trap-ws-size : N1
    interrupt-context : Context
    int-ws-base : Address
    int-ws-size : N1
    previous : Coroutine
    current-reason : Reason
    current-msg : Raw-Context
    msg-queue : Message*
    cr-stack : CR-Stack
    user-buffer : Message-Body
    buffer-empty : B
    handlers : Handlers
    raw-storage : Store
    init ...
end
```

The Idler co-routine is the "main" program upon start-up. The Idler co-routine is responsible for any initialisation and establishment of the other three co-routines which make up the monitor. The Monitor, Trap and Interrupt state are initialised by the Idler co-routine. The *handlers* component represents the interrupt/trap handler table. The *raw-storage* is the free pool from which space for namespace environments is allocated.

The PUT does not have its co-routine as part of Root state since this is maintained by the Target namespace which also maintains all entity information. Because the PUT co-routine is directly controlled by handlers in Target namespace, the co-routine state is maintained as a part of Target namespace rather than monitor global state.

## Models for Monitor Events

It will be remembered that the break event model recognises three types of break event that may occur. Of these, only the *LOCAL-BREAK* has any relevance with regard to a given machine—some machines explicitly support a break event exception, others allow the use of certain instructions that can be made to behave as break events for the purposes of the application. The *GROUP-BREAK* and *GLOBAL-BREAK* are *reasons* that cannot occur without Target namespace support.

```
types
    External-Breaks =
                    ⟨GROUP-BREAK⟩ |
                    ⟨GLOBAL-BREAK⟩;
    Break-Reason =
                 ⟨LOCAL-BREAK⟩ |
                 External-Breaks;
```

These are used to support the notion of *distributed breakpoints* across a group of distributed target machines.

a. The LOCAL-BREAK denotes the locally occurring break event that occurs when a break event is detected *local* to that target.

b. The GROUP-BREAK denotes the event derived from a local occurrence which signifies that a break event is propagated to other members of the group that exist on different machines.

c. The GLOBAL-BREAK denotes the event derived from a local occurrence which signifies that all entities within the PUT are to be halted. In a manner similar to the GROUP-BREAK, this means that the break event is propagated to all machines upon which the PUT is executing.

*Interrupts* are caused by the occurrence of some *external* event. In many cases this will be due to input or output. The model uses two kinds of interrupt, interrupts from communications channels and, if available, dynamic configuration interrupts which are caused by changes in hardware configuration. Interrupts from communication channels are typically the result of an input/output process completing. A dynamic configuration interrupt is an event that is caused by the insertion or removal of hardware devices from the target system. It is possible that the PUT may capture these events in addition to programming hardware that uses other available interrupts in the system.

```
types
    Interrupt-Reason =
                    ⟨COMMUNICATION⟩ |
                    ⟨DYNAMIC-ADD⟩ |
                    ⟨DYNAMIC-REMOVE⟩;
```

a. *COMMUNICATION* indicates the event that occurs when an interrupt driven communications device notifies the target processor that some action caused by the communication event needs servicing.

b. *DYNAMIC-ADD* indicates the (optional) event that occurs when the target hardware notifies the target system that some hardware device has been added to the target system.

c. *DYNAMIC-REMOVE* is the inverse of the *DYNAMIC-ADD* event and occurs when some hardware device has been removed from the target system.

*Traps* are events that occur as the result of code execution and are therefore synchronous to the PUT rather than some triggered external event. Traps are internal events in the sense that the execution context prior to the trap is the cause of the trap. Trap reasons can vary considerably between target configurations. An example of typical traps are illegal instructions, bus errors and divisions by zero. It is possible that the PUT may handle its own traps but for those it does not, the Trap co-routine and subsequently the Monitor co-routine *must* handle them.

```
types
    Trap-Reason =
              ⟨TARGET-DEPENDENT-TRAP⟩;
```

The *Reason* is the union of all the types discussed above. This type will be larger than the available reasons a target system has because some reasons are *constructed* rather than occurring natively as part of target hardware operation. This is especially the case with *external break events.*

```
types
    Reason =
        ⟨NOT-A-REASON⟩ | Break-Reason | Interrupt-Reason | Trap-Reason;

    Target-Trap =
            token;
```

### Breakpoint Events

In addition to local breakpoints, it will be remembered that group and global breakpoints are supported to allow the human user to specify distributed breakpoints across a group of target machines. A mapping from external breakpoint message types to monitor native form is required:—

types
*Break-Identity* ::
*kind* : *Breakpoint-Kind*
*id* : token;

*External-Break-Data* =
*Break-Identity*
inv *b-id* $\triangleq$
let *mk-Break-Identity(kind,_)* = *b-id* in
*kind* $\in$ *External-Breaks;*

An external break message is a *Msg-Element* whose *Msg-Kind* is EXTERNAL-BREAK. It is received from an external source such as another *monitor.* If the break message received from another monitor then it is a *group* breakpoint. If it is a break message received from the *debug agent* then it is a *global* breakpoint. The message data consists of the *Break-Identity* that contains information regarding the *kind* of break event (group or global) and the *name* of the break event. How this information is used when such messages arrive was described in the break event handler specifications given in Chapter 5.

functions
*map-external-break* : *Breakpoint-Kind* $\rightarrow$ *External-Breaks*
*map-external-break(ext-brk-value)* $\triangleq$
1. (
2. cases *ext-brk-value:*
3. $\langle GROUP \rangle \rightarrow$
4. $\langle GROUP\text{-}BREAK \rangle$,
5. $\langle GLOBAL \rangle \rightarrow$
6. $\langle GLOBAL\text{-}BREAK \rangle$
7. end
8. )

## Low-Level Model Types

The following type definitions model program activations as *co-routines.* A co-routine exhibits the following behaviour:—

a. It may have its execution suspended by either explicitly performing a transfer to another co-routine or by yielding control implicitly when an interrupt occurs.

b. Execution is resumed at the point at which suspension took place at some later time.

For a co-routine to resume execution at any point requires that a "frozen" state of activation at the point of suspension and transfer be maintained, this is known as a *context* of a co-routine. Within the context there exists a *closure*, a snapshot of the register contents at the point where the co-routine transfer of control occurred. Because there are many different processor architectures present, the context and closure maintained will differ on a per target basis.

In addition to co-routine context and closure, a co-routine also must maintain its own stack used to store calling frame activation and local variables. For this purpose a co-routine's entire context and storage area are part of a co-routine's *workspace.*

```
types
   Coroutine =
            ⟨IDLER⟩ | ⟨TRAP⟩ | ⟨INTERRUPT⟩ |
            ⟨MONITOR⟩ | ⟨PROGRAM-UNDER-TEST⟩;

   Context ::
         what : Address – the workspace
         where : Address – the current code position
         entity : Coroutine
         machine-state : Address; – the register dump position
```

The *Context* type is the abstracted model context used by the monitor which is a mapping from *Raw-Context.*

```
types
   Raw-Context ::
            tgt-trap : Target-Trap
            tgt-context : Coroutine
            machine-state-ptr : [token]
            dyn-space : [token]
            code-locn : [token];
```

The *Raw-Context* type is used to model the real context of a co-routine when halted due to the occurrence of some event.

```
types
   Trap-Msg ::
          interrupted : Context
          reason-code : Reason;

   Breakpt-Msg ::
            trapmsg : Trap-Msg
            b-id : [Break-Identity]
               inv brk-msg ≜
                  let mk-Breakpt-Msg(trap-msg,_) = brk-msg in
                     let mk-Trap-Msg(_,reason) = trap-msg in
                        reason ∈ {⟨LOCAL-BREAK⟩,
                                  ⟨GROUP-BREAK⟩,
                                  ⟨GLOBAL-BREAK⟩};
```

The *Breakpt-Msg* is a special case of the more general *Trap-Msg* type. This special type is defined in order to specify that *all* breakpoints behave the same regardless of the source, be it locally caused or externally caused through a message sent by another monitor or debug agent. The *b-id* field contains the identity of the breakpoint descriptor used for identifying a group breakpoint. It is not used when a *LOCAL-BREAK* or *GLOBAL-BREAK* message is sent to Target namespace and is therefore set to nil.

## Low-Level Operations

The following operations manipulate various forms of saved machine context that become available when traps and interrupts occur. This section describes the operations needed to support the trap co-routine, the operations required for the interrupt co-routine are similar in structure and are discussed later with respect to that co-routine.

The *get-trap-reason* operation returns the cause of the interruption and the context of the last interrupted co-routine. If an external breakpoint has occurred, the information required for the

breakpoint handler has already been pushed onto the message queue for processing by the monitor co-routine by the time that this operation is invoked. The trap co-routine therefore does *not* need to know anything about the break identity. All that is required by the *invoker* of *get-trap-reason* is the reason code.

operations
*get-trap-reason() trap : Trap-Msg*

The *get-trap-reason* operation uses the following operations to support the construction of event messages to be dispatched to the monitor queue.

The functions, *map-data, map-code* and *map-closure* extract the current stack pointer, program counter and machine dump state pointer from a *Raw-Context* available after an interrupt or trap has taken place. These values are used to generate a *Context* used by the monitor to provide support for the *Entity* namespaces subordinate to Target namespace. The closure includes the interrupted *program-counter* and *stack-pointer* values acquired by *map-code* and *map-data*.

functions
*target-dependent-interrupt-handler() r : Raw-Context*
post
  *r* = target-dependent

*target-dependent-reason(raw-reason : Target-Trap) reason : Reason*
post
  *reason* = target-dependent

*map-data(context-data-ptr :* token*) ws-ptr : Address*
post
  *ws-ptr* = target-dependent

*map-code(context-code-ptr :* token*) pc : Address*
post
  *pc* = target-dependent

*map-closure(context-closure-ptr :* token*) closure : Address*
post
  *closure* = target-dependent

NOTE Most, if not all of the above operations will require only as few as one machine instruction to a dozen or so machine instructions.

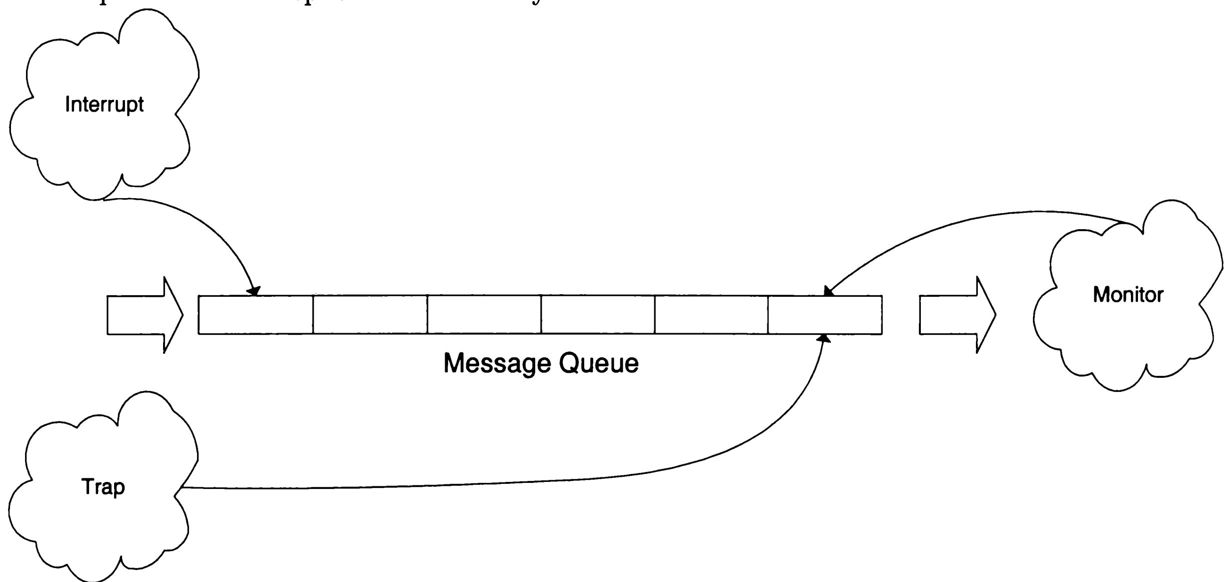
The Interrupt co-routine requires an operation, *get-interrupt-reason*, similar to *get-trap-reason* described earlier. When an interrupt occurs, the *Context* is extracted from the entity's *Raw-Context* which was interrupted. Typically this context will be used to return control to the interrupted entity when the interrupt co-routine has completed its tasks.

In the case of an external breakpoint message arriving, the *Breakpt-Msg* is constructed as a product of the *GROUP-BREAK* or *GLOBAL-BREAK* kind and the current context of the interrupted entity. This message is placed on the queue using *Push* and the interrupt handler transfers control to the trap co-routine. In this manner, external breakpoints appear to the monitor as being no different to local breakpoint occurrences. The actual breakpoint *handling* within Target namespace is different, given the source of the breakpoint as was described in Chapter 5.

### The Monitor Co-routine

The Monitor co-routine consumes and processes messages enqueued by Interrupt and Trap co-routines. It is an event loop. Messages are taken from the queue that "connects" Interrupt and Trap

co-routines with the Monitor co-routine. The Monitor co-routine dispatcher could be considered as the "top" of the namespace tree hierarchy.




Traps are local event occurrences that are prioritised ahead of other external events. Trap messages are therefore enqueued at the head of the queue whereas interrupt events result in messages that are enqueued at the tail of the queue.

Fig. 6.3 *The Trap, Interrupt and Monitor Co-routine relationship*

The following pseudo-code is a skeleton of the essential operations that need to be performed by the Monitor co-routine.

```
Monitor-Coroutine()
begin
    initialise monitor
    loop
        get message from queue
        – transfer to idler if there is no connection OR
        – there are no messages to be processed
        if connected then process-message()
    end
end

process-message()
begin
    case of
    NAME: dispatch to root namespace finder
    RESPONSE-DATA: reply to debug agent
    ERROR-DATA: reply to debug agent
    Other: reply to debug agent with error
end
```

If there is no connection, the monitor waits for an incoming message or event by transferring to the Idler co-routine. The Idler co-routine *resumes* the co-routine that transferred to the Idler after an event (be it an interrupt or trap) has occurred. Even in the case where the monitor event loop is executing, if there are no messages available on the queue to be processed, the *Pop* message operation will transfer to the Idler co-routine until such time as a message arrives for processing by the Monitor co-routine.

### The Trap Co-routine

Traps may occur for a variety of reasons, none of which the PUT may be able to handle. The occurrence of a trap causes the trap handler to be invoked, a message constructed and enqueued so that the monitor co-routine can then be transferred to.

The trap co-routine attempts to determine the reason for the trap, where the trap occurred and the entity that was executing when the trap occurred. A *Trap-Msg* is constructed and enqueued in such a way that the first message to be processed by the monitor co-routine will be the trap message. As long as the transfer to the monitor takes place after the trap occurs, real-time events such as breakpoints will be handled *immediately* after they occur.

```
Trap-Coroutine()
begin
  set initial co-routine to transfer to Idler
  attach all trap sources to the trap co-routine
  loop
    transfer to last interrupted co-routine – initially Idler...
    get trap reason
    if connected to debug agent then
      trap-dispatch()
    else
      ignore and continue
  end
end

trap-dispatch()
begin
  retrieve closure from previously interrupted co-routine
  case of
  GROUP-BREAK, GLOBAL-BREAK: transfer to Monitor co-routine
  LOCAL-BREAK: Push handle break message onto queue
           Transfer to Monitor co-routine
  Others: Push error message onto queue
        Transfer to Monitor co-routine
end
```

### The Interrupt Co-routine

*Interrupts* differ from *traps* in that their occurrence does not necessarily cause a transfer of control to the Monitor co-routine. Generally an Interrupt co-routine will handle the event, possibly enqueue a message onto the message queue and then return control to the entity which was previously interrupted. This is to minimise monitor invasiveness as much as possible. The only messages that cause a transfer of control away from the interrupted co-routine are the external breakpoint messages.

The *Interrupt Co-routine* consists of the following steps:—

(1) Wait until an interrupt occurs.

(2) Extract co-routine information from interrupted context.

(3) Handle interrupt based on the type of interrupt; communications, dynamic add or dynamic remove. Unhandled interrupts without connected session are skipped, otherwise an error message is returned to the debug agent.

(4) Loop back to step 3.

```
Interrupt-Coroutine()
begin
  set initial co-routine to transfer to Idler
  attach all interrupt sources to the interrupt co-routine
  loop
    transfer to last interrupted co-routine – initially Idler...
    get interrupt reason
    case of
    Communication:do inbound message processing
    Dynamic-Add:do hardware addition processing
    Dynamic-Remove:do hardware removal processing
    Others:if connected then send error message
         else do nothing
    set to last interrupted co-routine
  end
end
```

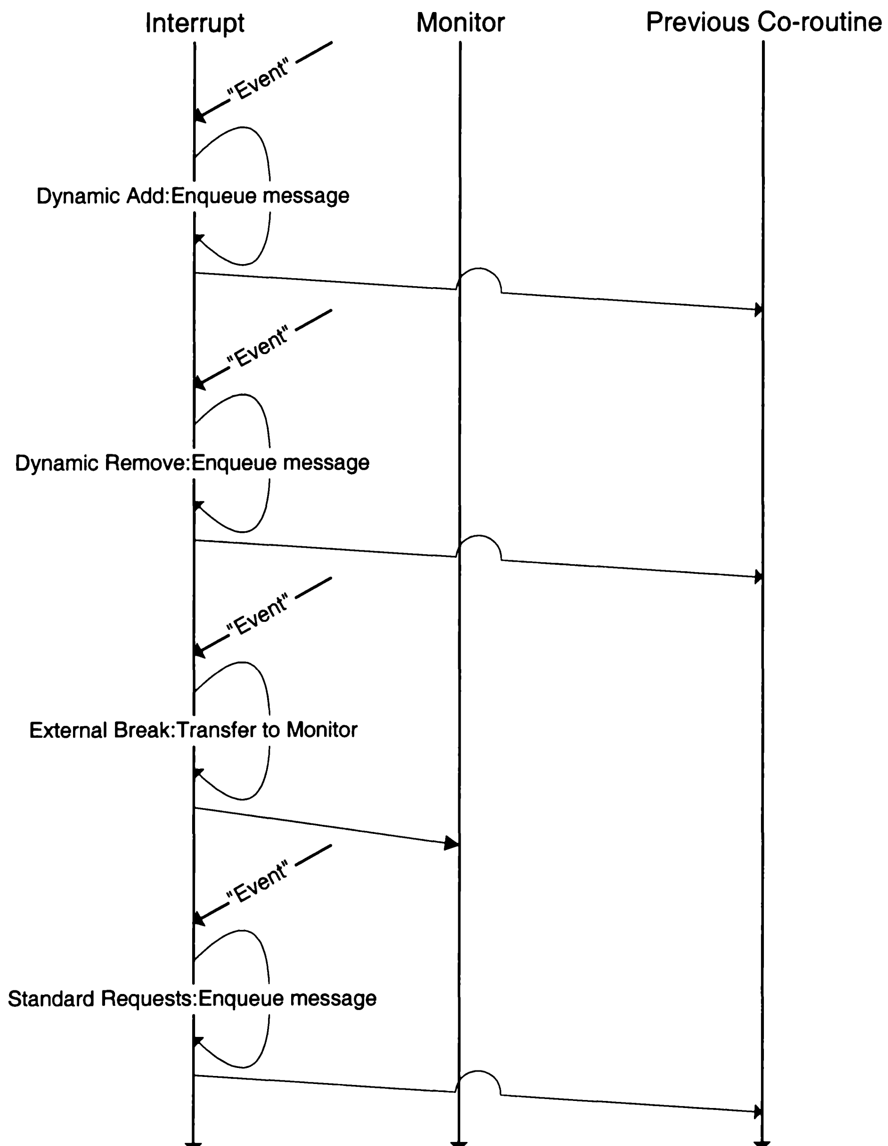



Fig. 6.4 *Interrupt Co-routine timing and sequencing*

### Interrupt Side-Effects

Enqueued messages are not processed until the Monitor co-routine is resumed. The monitor can only execute when a trap occurs or the PUT terminates (normally or abnormally). This implies that on-the-fly namespace handler processing is not possible. The reasons behind this behaviour are to do with minimising the *Probe Effect*. Any monitoring done on-the-fly would cause the performance of the PUT to be perturbed. This could potentially mask any race conditions that the debugger user is attempting to isolate!

On-the-fly namespace handler processing *could* be done by allowing incoming messages to be treated as traps, much in the manner that external breakpoints are treated as traps even though they were effectively interrupt caused. This does, however, have the major cost of perturbing performance of the PUT. Another method which could be used involves the use of breakpoint predicates to send information back to the debug agent. Since breakpoint predicates are user definable, as breakpoints are set, the breakpoint predicate can be defined in such a manner that a dump of memory or register contents could be sent back to the debugger. This method does not decrease invasiveness but does take advantage of an existing mechanism which simplifies a potential implementation of the technique down to setting a breakpoint and an appropriately devised predicate. While the alternative on-the-fly namespace handler processing is available if desired, it will not be considered further in this thesis.

#### The Idler Co-routine

The *Idler* co-routine is used to bootstrap the monitor in addition to the fact that it is an important "glue" operation. The Idler executes when no other co-routine can run:—

a. When there are no messages in the queue for the monitor to process, the *Pop* operation transfers to the Idler until such time as there is a message to process. The Idler will execute until such time that a message arrives after which it will return control to the Monitor co-routine so that the message can be processed.

b. The Idler co-routine also executes when there is no PUT executing or no session connection.

c. The Idler is also used by *User Services*. When the PUT invokes the primitive input facilities provided by User Services, the input handler transfers to the Idler when message buffers are empty. The wait is satisfied when user data arrives.

```
operations
Idler-Coroutine : () -o-> ()
Idler-Coroutine() ≜
1.  (
2.     idler-context := bootstrap-idler();
3.     initialise-bootstate();
4.     initialise-coroutines();
5.     enable-interrupts();
6.     while true do
7.     (
8.        let mk-Env(_,root-map,_,_) = ρroot in
9.           let mk-Env(ext-ws,_,_,_) = root-map(⟨EXTERNAL⟩) in
10.             let mk-Workspace(_,_,ext-state) = ext-ws in
11.                let mk-External-State(_,_,_,_,_,_,_,_,idle-op,_) = ext-state in
12.                   def dummy = transfer-to-coroutine(idle-op(idler-context.entity)) in
13.                      skip
14.    )
15. )
```

The *bootstrap-idler, initialise-bootstate, initialise-coroutines* and *enable-interrupts* operations are used to establish the monitor. The *bootstrap-idler* operation performs implementation-dependent operations that establish the Idler as a co-routine upon start-up. When *initialise-bootstate* is invoked, the state of the monitor global state is established. The *initialise-coroutines* operation is used establish the other co-routines of which monitor is composed. The default channel of communication is also enabled. The last phase of this initialisation is a target-dependent means of enabling interrupts to occur so that messages from the debug agent or dynamic configuration events can be actioned upon when they occur. These operations are defined further in Chapter 8.

The role of the *idle-op* operation within the *Idler-Coroutine* depends upon whether the current communication device channel is interrupt-driven or polled. If the device is interrupt-driven the *idle-op* operation becomes a *halt machine.* This is analogous to a machine STOP or HALT instruction being executed in the instruction stream, no further activity takes place until an interrupt occurs. An asynchronous event causes a "wake-up" to take place and a resumption of machine processing. In the case of a polled device the *idle-op* becomes a *poll* operation with the device polling a message arrival. The poll operation becomes the message receiving operation for the device driver, providing a "pseudo-interrupt" when a message is complete. When a complete message is received, processing takes place by transferring to the *Interrupt-Coroutine* where the message is received in the same manner as if an interrupt driven device were present—this also implies that it is up to the poller to make sure that the message arrival appears to be an interrupt, making sure that the *current-reason* reflects that a communication interrupt has taken place. This treatment of message arrival as an interrupt regardless of the fact that the channel is polled or interrupt-driven ensures consistency and simplicity, an immediate comparison can be made to the treatment of external breakpoint messages as if they were local breakpoints as indicated in the description of the *trap* and *interrupt co-routines.*

## The Global Message Queue

The user application and local event/message sources are producers of messages. The monitor consumes and processes these messages. The monitor software (the namespace environments and their respective handlers) may even produce and enqueue these messages for future processing by the monitor. Communication between the three entities takes place via the global message queue. Figure 6.5 illustrates this relationship.

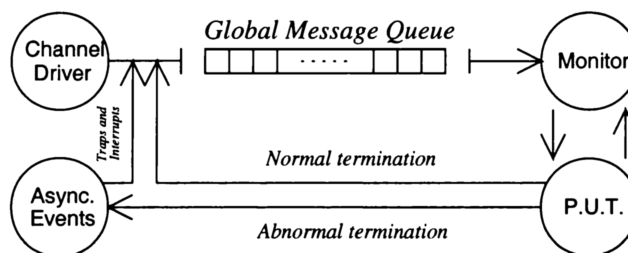



Fig. 6.5 *Message queue and monitor software relationship*

While the monitor co-routine processes messages in FIFO order, messages can be placed at either the *head* or *tail* of the queue. Messages arising from trap processing are placed at the head of the queue, a so called *Push* operation. Messages arising from interrupt processing with the exception of external breakpoint messages (which are treated as traps) are placed at the tail of the queue using the *Insert* operation.

Although a fully refined implementation of the message and its operations is not given here, one of the issues, that of a full message queue, does need consideration. In such a situation, external

messages from debug agents should be the first to be dropped as such messages can always be re-transmitted. Messages that are sent from other monitors cannot be rejected since the monitors that sent the messages cannot time-out and resend, as a result, any such message can never be rejected. The message queue must be "large enough" to buffer for situations where there are many messages arriving from other monitors such as when a global or group break occurs. Local messages cannot be dropped from the message queue as the events which caused those messages cannot be duplicated. The simple rule in this case is:—

> *Events such as traps and break events have priority over debug agent initiated messages. Events are not repeatable and must be captured as they arise, messages from the debug agent can always be re-transmitted at some time in the future.*

```
operations
Pop : () -o-> Message
Pop() △
1.  (
2.     if msg-queue = [] then
3.     (
4.        def dummy = transfer-to-coroutine(idler-context.entity) in
5.           skip;
6.        return Pop()
7.     )
8.     else
9.        let monitor-msg = hd msg-queue in
10.       (
11.          msg-queue := tl msg-queue;
12.          return monitor-msg
13.       )
14. )
```

The *Pop* operation removes a message from the head of the queue and is for use by the Monitor co-routine. The Idler co-routine is resumed when there are no messages available to be processed by the monitor. In the case of an interrupt driven communications channel, the system will halt within the Idler co-routine. Any interrupts occurring after the halt or busy wait will cause the a transfer to the Interrupt co-routine. The Interrupt co-routine will then enqueue any messages and return to what transferred it—the Idler co-routine. Having satisfied the halt, the Idler co-routine can then transfer control to the co-routine that originally transferred to it—the Monitor co-routine. The monitor then proceeds with at least *one* message that it can process.

In the case where a communication channel is driven via polled input-output, the situation is similar but requires the driver to perform some operations which would occur automatically with an interrupt driven device. An empty message queue will cause control to be transferred from monitor to idler. This time *idle-op* becomes a poll operation responsible for acquiring the message packet. When the message is completely received, the poll operation transfers control to the Interrupt co-routine which may process the message immediately if it is destined for External namespace or enqueue the message received if it is destined for the monitor. When the Interrupt co-routine has finished with the message it can transfer back to the poller which then transfers back to the Monitor co-routine for the same net effect as above.

The *Push* and *Insert* operations place a message at the head of the queue and tail of the queue respectively.

operations
*Push(msg : Message)*
ext wr *msg-queue : Message**
post
$msg\text{-}queue = msg \frown \overleftarrow{msg\text{-}queue}$

*Insert(msg : Message)*
ext wr *msg-queue : Message**
post
$msg\text{-}queue = \overleftarrow{msg\text{-}queue} \frown msg$

### Messages Received from the Outside World

The *Communication-Handler* is only invoked when the interrupt co-routine is active and the reason for the interrupt is a *Communication.* The primary function of the handler is to receive a message that has arrived from either another monitor or the debug agent. The message received is dispatched to an appropriate dispatch handler depending on the type of the leading *Msg-Element.*

The contents of a message received are subject to filtering done by the *retrieval* functions to be described in Chapter 9. Messages received include EXTERNAL-BREAK, EXTERNAL-NAME, USER-DATA, NAME and ERROR messages resulting from bad retrievals. Errors that result from malformed or incorrect requests cause *immediate* replies to the debug agent.

The *External-Name-Dispatch* is invoked by the *Communication-Handler* when the message tag is EXTERNAL-NAME. This behaves in a similar manner to the monitor co-routine which invokes the Root level finder. In the case of the External namespace however, the External namespace finder is invoked. There is *no* enqueueing of messages to be processed by the monitor co-routine, messages to External namespace are handled as they arrive. Such messages include those to open and close sessions between monitor and debug agent.

### External Breakpoint Messages

There are two kinds of external breakpoint messages, group breakpoints and global breakpoints. As the occurrence of such a message requires breakpoint handling, the *External-Break-Dispatch* handler transfers control to the Trap co-routine causing an eventual transfer of control to the Monitor co-routine. The behaviour resulting from an external breakpoint message is exceptional in that it is the only case where an interrupt causes a transfer to the Monitor co-routine (albeit indirectly via the Trap co-routine). This designed behaviour makes external breakpoint occurrences appear to be like local breakpoints.

The concept of external breakpoints is a software solution where no equivalent solution exists in hardware. Events generated by a given CPU are not *directly* propagated to other external processors so the above software solution was used.

Only the *group* breakpoint has an associated break identity. The *global* breakpoint has no associated break identity, such messages stop *all* activity of the PUT on the target machine. This is subject to the completion of any finalisation that needs to be completed to bring the PUT to a safe halt as described earlier.

### User Data Messages

The *User-Data-Dispatch* handler is used to extract user data service packets from the input stream. The model described here does not buffer any user data packets. The *buffer-empty* Boolean

is used to determine whether a message can be consumed. When set to false, the buffer can be retrieved by the user service library function, *User-Get*. When set to true, the User-Get operation must wait for a USER-DATA message to arrive.

operations
*User-Data-Dispatch(interrupted : Coroutine, msg : Message) res : Coroutine*
ext wr *buffer-empty* : **B**
ext wr *user-buffer : Message-Body*
post
*1.* (*buffer-empty* = false) $\wedge$
*2.* (*user-buffer = Message-Value(msg(1))*) $\wedge$
*3.* (*res = interrupted*)

## Namespace Engine Messages

The *Name-Dispatch* handler enqueues a message for future processing by the Monitor co-routine. If a debug agent tries to send messages to the Monitor co-routine without a valid session context, the request will be *ignored* otherwise the message will be inserted onto the message queue and a transfer to the Monitor co-routine will made if Idler co-routine was previously executing.

The Interrupt co-routine is activated when an interrupt occurs. Tagged messages arriving via the communications device, with the exception of EXTERNAL-NAME and EXTERNAL-BREAK, cause a return to the last interrupted co-routine. This is what distinguishes a Trap co-routine from an Interrupt co-routine. Traps *always* cause control to be transferred to the Monitor co-routine.

External namespace which contains handlers used by Interrupt and Trap co-routines can be viewed as a separate state machine which accepts and filters messages destined for either External namespace or "internal" namespaces. The Interrupt co-routine filters requests. Messages which have an *EXTERNAL-NAME* tag are processed immediately by External namespace, *NAME* tags are enqueued for processing by the monitor co-routine.

## Dynamic Configuration of Hardware

If hardware mechanisms exist to allow notification of changes of hardware configuration via an interrupt there will be interrupt handlers that allow the messages to be sent to the debug agent.

The *Register-Hardware* interrupt handler determines the device parameters upon device insertion and then produces two messages which it enqueues to the Monitor co-routine. The first message is used to provide hardware information to the debug agent that may assist it in determining a symbolic description of the namespace for use by the debugger. The second message adds the hardware to device namespace. The eventual reply from the invocation of *Add-Hardware* is used as an interlock to ensure that the agent does not attempt to use the namespace without first making sure that the namespace added actually exists. Due to lack of resources the addition of a namespace may fail. In this case, the debug agent cannot access the namespace even though the hardware the namespace represents may physically exist.

The analogous *Deregister-Hardware* is provided for removal of the hardware device from the system. As the hardware identity is used as the name of the device, its removal is accomplished by removing the namespace using the *Remove-Name* handler of Device namespace.

The dynamic configuration handling presented above assumes that the backplane hardware used for expanding a system is capable of notifying software through an interrupt mechanism when dynamic configuration of hardware takes place. This may not always be possible. Target hardware may not support such a feature. In the case where no notification is possible or even worse still,

only static configuration is allowed, notification to the monitor can be *debug agent initiated.* As the namespace model permits the debug agent to add namespaces and handlers as it sees fit, it makes sense that where interrupt-driven dynamic configuration/notification are not available, the debug agent can "fill in the gaps" where necessary, allowing both debug agent and monitor to keep their configurations synchronised.

What constitutes useful hardware information to convey to the debug agent when such a dynamic configuration takes place? Given that a namespace is added when the hardware configuration changes, the namespace *must* have a unique name. Each hardware object added must have some physical presence on the target machine—for example, a card filling a slot on a backplane. A possible unique naming discipline then would be to use a *slot-id* mechanism. Once inserted into a slot, further configuration has to take place—where does the card map into the system address space? How large a chunk of the address space does the card take up? Other, purely informative attributes are also useful. What type of card is it? Who manufactured it? It is this information that is provided by the functions *get-new-added-hw-id, get-hw-info* and *get-new-removed-hw-id.* The need for information attributes such as card type and card manufacturer is to assist the debug agent in determining the correct symbolic description that the debug agent may use as a namespace descriptor. The following type definitions from Chapter 5 are repeated here for clarity.

types
  *Slot-Id* =
    token

  *Port* ::
    *base* : *Address*
    *size* : $\mathbf{N}_1$ ;

The *Port* structure models hardware that may use the address space to which it is mapped, in a sparse manner. For example, a communications device might map into memory in such a manner where each register is followed by a hole in the address space which maps to a void area.

  *Hardware-Resource* ::
      *ports* : $Port^+$
      *product-Id* : [token]
      *manufacturer-Id* : [token]
      *unit-serial* : [token]
      *comment* : [$\text{char}^+$]

operations
*get-hw-info() hw-info* : *Hardware-Resource*
post
  *hw-info* = undefined

*get-new-added-hw-id() slot-name* : *Slot-Id*
post
  *slot-name* = undefined

*get-new-removed-hw-id() slot-name* : *Slot-Id*
post
  *slot-name* = undefined

Each of the above functions retrieves information about the hardware resource in a manner that is both device-dependent and target-dependent. The hardware attributes suggested above in the type declarations are the minimum required for *unique* identification of a hardware object. The type *Slot-Id* modelled as a token is useful in the sense that it provides a unique name to the device inserted.

## Summary

The namespace model cannot exist without an environment that is capable of driving the execution of namespace handlers. This chapter discussed the underlying monitor framework whose primary purpose is to provide a message and event stream for consumption by the namespace model.

The monitor framework consists of four co-routines; the Idler, Monitor, Traps and Interrupts. It is primarily the Monitor co-routine that passes each message to the namespace handler dispatch mechanism. The other co-routines are responsible for capturing messages and events and ensuring their propagation to the Monitor co-routine.

Because the namespace model is dynamically modifiable, the framework must provide operations that allow for the management of storage used for the creation of namespace environments and co-routines.

While this chapter has described the mechanisms for message processing it has neither considered the issues of acquiring and sending messages nor has it considered message representation. These issues are discussed further in the following chapters.

# Chapter 7

# Feeding the Model

This chapter describes the design of the input-output system used by the monitor to communicate with the debug agent and other monitors.

Because the input-output system is like any other service supported by the monitor, it too is modelled using the namespace abstraction. However, the services provided by the input-output system are not directly accessible via the debug agent. In fact, the services themselves are only ever implicitly invoked whenever debug agent–monitor input-output is performed.

### Communications Namespace—Channel drivers

Communications namespace is much like the Entity namespaces described in Chapter 5. In the same way that Target namespace managed entity namespaces, it is the responsibility of External namespace to maintain Communication (driver) namespaces and a current channel of communication. The drivers used to communicate between the debug agent and monitor server are just subordinate namespace environments below External namespace.

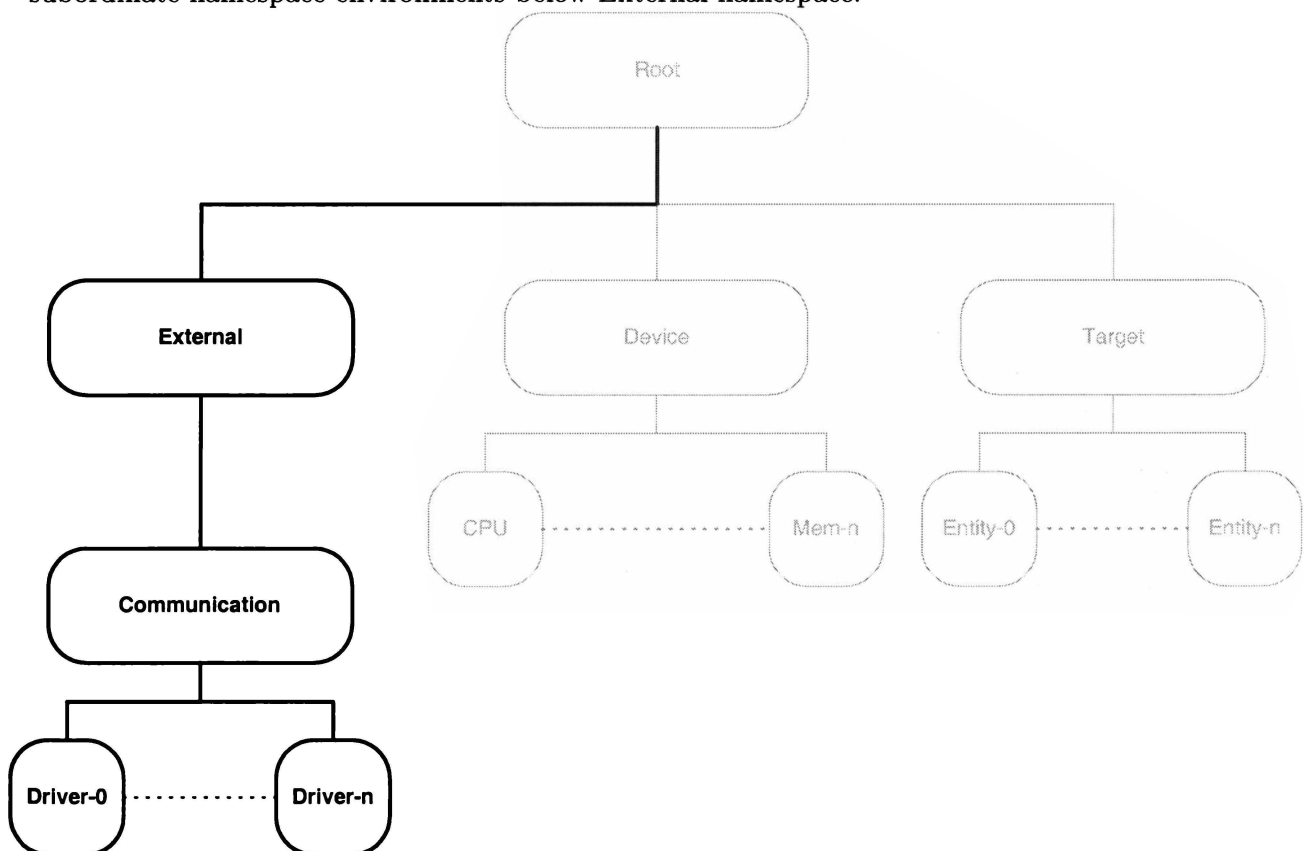



Fig. 7.1 *External Namespace and the relationship to Communications Namespaces*

External namespace provides device driver access to channels used by the monitor server to communicate with the debug agent. The monitor software may have channel drivers for one or

more devices that are known to it. There must be at least *one* channel device present for useful debugging to take place. To achieve full distributed debugger functionality, it is necessary to have at least two channel drivers for the devices concerned, one for channel used by the PUT and the other for use by the monitor. These channels are independent, the monitor only receives messages sent to it by the debug agent or other monitors. The PUT sends and receives messages along the channel it uses to communicate with other target machines on which the PUT resides.

The reasons for the monitor design including multiple channels of communication through driver namespaces subordinate External namespace is to support the development of software that requires the use of communications hardware. A multi-device communication system allows the development of software that can use the communications devices while still allowing (provided there are available device drivers and physical devices) monitor communication to take place. The PUT should be allowed to use *any* communication device to support its functionality. If the monitor can support multiple device drivers, the monitor can always switch to a communications device which the PUT is not using.

Each device driver is an environment, a namespace within External namespace. A driver has only one Work-Handler operation that it *must* provide—*Driver-Initialise.* The *Driver-Initialise* handler is usually invoked by *Check-Driver.* The *Add-Hardware* handler of Device namespace invokes *Check-Driver.* If a device driver exists for a particular piece of hardware just added to the system then (provided the driver exists) initialisation of the driver takes place.

*All* driver operations are encapsulated within the *Driver-Initialise* Work-Handler. Such initialisation establishes two things:—

a. Initialisation of the *port-map*, the address mapping where the physical device registers exist in the memory map. These registers are used to establish the hardware settings required to drive the device.

b. Initialisation of device driver *entry points.* These represent the operations performed by a given device to implement communications functionality:—

(1) The *driver-init* operation tests whether a given device can be initialised and therefore possibly used by the monitor to provide a communications channel to the debug agent. It is this operation that determines the driver entry points and initialises the *Driver-Table* within the workspace.

(2) The *open-read* operation initialises the channel for reading.

(3) The *open-write* operation initialises the channel for writing. The channel used for receiving messages may not necessarily be the one used for sending messages. This operation returns the monitor address that is to be used by the agent if it is to send messages to the monitor.

(4) The *get-msg* operation returns a message received by that channel and the address of the sender of the message.

(5) The *put-msg* operation takes a message and an address to send to and sends that message to the destination given. The destination may be the debug agent or possibly another monitor.

(6) The *close-read* operation resets a channel, disabling only the ability to receive messages from that channel.

(7) The *close-write* operation resets a channel, disabling only the ability to send messages down that channel.

(8) The *poll* operation is used to collect an octet stream from some device until an entire message has been collected. If a device receives messages by the use of polled input-output this operation is used by the idler co-routine to collect messages while the monitor co-routine is effectively in control.

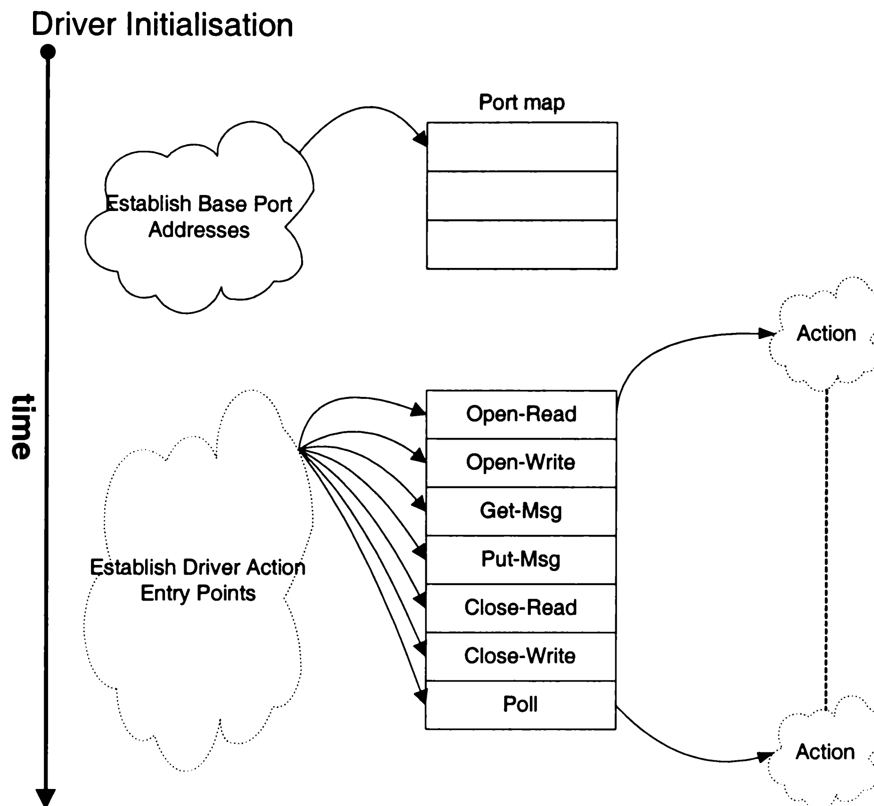



Fig. 7.2 *Initialising driver operations with Driver-Initialise*

Device drivers may be implemented using polled, interrupt-driven or direct memory access driven input-output. There is no restriction other than the fact that polled device drivers result in a loss of debugging functionality provided by the monitor—messages cannot be received while the PUT is executing. Messages can only be received while the the monitor is active, this means that external breakpoints cannot be caught by the monitor as the polled read operation may not be active when the message arrives at the monitor.

A polled device driver does not attach to an interrupt source. A polled receiver attaches itself to the *poll* operation within the Idler co-routine. If the communications channel is interrupt-driven then this operation is a *halt processor* instruction. The Idler is responsible for performing polling operations when no messages are on the queue. The polled device might drive something like a serial line, in this case the driver *accumulates* a *Raw-Message* until complete and then sets the *current-reason* to appear as if a communication channel interrupt has occurred. The poller then transfers control to the interrupt co-routine which handles the message appropriately, enqueueing it for monitor usage if necessary. When the interrupt co-routine is complete it transfers control back to the Idler (as per Interrupt co-routine behaviour). When the message has been received the Idler resumes the co-routine that originally transferred to it—the Monitor. As the Monitor has a message to operate on, it can continue processing the namespace message.

Interrupt-driven devices are executed as part of the Interrupt co-routine. If the reason determined is the result of an external communication, the *Receive* which is a part of External namespace will be invoked and the message received, the behaviour of the Interrupt co-routine is such that

the co-routine active immediately before the interrupt will be resumed when interrupt processing is complete. This means that messages *can* be received while the PUT is executing.

The device driver interface provided by each driver namespace within External namespace is simple because there is no need for multiplexing messages received, all messages that arrive down the channel currently in use by the monitor are messages that are *only* for the monitor. The only structure required to support driver initialisation is the *Port* structure described in Device namespace. The *Port* structure describes an *address map* of a particular hardware device. When hardware configures dynamically (or even statically) the mapping that takes place may vary considerably over a number of system configurations. This means that device drivers cannot assume that the registers they use to initialise a channel device are in a particular block of memory addresses. Device drivers have to be written in such a way that the code does not lock a driver to a fixed set of addresses. The state of each driver namespace has a *Port* structure so that if address mappings change, the driver can still be initialised regardless of the remapping.

types

*Init-Result* =
  **B**;

*Open-Result* =
  **B**;

*Close-Result* =
  **B**;

*Read-Proc* =
  *()* $\rightarrow$ *Raw-Message* $\times$ *Machine-Address;*

*Write-Proc* =
  *Raw-Message* $\times$ *Machine-Address* $\rightarrow$ **B**;

*Driver-Table* ::
  *open-read : ()* $\rightarrow$ *Open-Result*
  *open-write : ()* $\rightarrow$ *Open-Result* $\times$ *Machine-Address*
  *close-read : ()* $\rightarrow$ *Close-Result*
  *close-write : ()* $\rightarrow$ *Close-Result*
  *get-msg : Read-Proc*
  *put-msg : Write-Proc*
  *poller : [Coroutine* $\rightarrow$ *Coroutine];*

Annotate

If *poller* is nil then the device does not use polled input-output.

End Annotation

types

*Driver-State* ::
  *port-map : Port*$^+$
  *dev-tab : Driver-Table;*

state $\rho_{driver}$ of
  *drv-ws : Workspace*
  *drv-map : Map*
  *my-name : Name*
  *finder : Env-Handler*
  init ...; – This environment is initialised as part of Monitor global state. Refer to Annex B.
end

The operations referenced in the *Driver-Table* implement actual driver behaviour to enable or disable the channels of that device. As with all namespaces, driver namespaces can be added

or removed dynamically. Adding a device driver is a matter of adding a *Driver-Initialise* Work-Handler to that namespace. The actual entry point to the *Driver-Initialise* operation ensures that the table is initialised to correctly reflect the entry points of driver functions.

The *Driver-Initialise* operation is generally not explicitly invoked by a message from the debug agent although it is not prevented from doing so. If the target hardware supports dynamic configuration then the addition of that device namespace using *Add-Hardware* ensures that (provided a driver namespace exists) *Check-Driver* is invoked and the driver namespace is initialised for possible future use.

Device drivers share a common interface but ultimately the operations used to manipulate the hardware are *device-dependent.* The following definitions are given for the purposes of completeness and represent device-dependent driver operations required in the implementation of a device driver that the *Driver-Initialise* handler establishes as entries in the *Driver-Table.*

**operations**
*open-read* : () $\xrightarrow{\circ}$ *Open-Result*
*open-read*() $\triangleq$
(
  **device-dependent**
)

*open-write* : () $\xrightarrow{\circ}$ *Open-Result* × *Machine-Address*
*open-write*() $\triangleq$
(
  **device-dependent**
)
*close-read* : () $\xrightarrow{\circ}$ *Close-Result*
*close-read*() $\triangleq$
(
  **device-dependent**
)

*close-write* : () $\xrightarrow{\circ}$ *Close-Result*
*close-write*() $\triangleq$
(
  **device-dependent**
)

*get-message* : *Read-Proc*
*get-message*() $\triangleq$
(
  **device-dependent**
)

*put-message* : *Write-Proc*
*put-message*(*msg*) $\triangleq$
(
  **device-dependent**
)

*poll* : *Coroutine* $\xrightarrow{\circ}$ *Coroutine*
*poll*(*the-monitor*) $\triangleq$
(
  **device-dependent**
)

The *put-message* and *get-message* operations send and receive an octet stream to and from the actual communications medium. The steps taken to perform *retrieval* and *concretion* of the message stream are performed by the handlers of External namespace or possibly the driver operations

themselves. These refinement issues are irrelevant from the model's point of view but are discussed further in Chapter 9.

Device driver initialisation is performed by *driver-init*. The operations used to establish the initial state of a device are device independent but the result is a table of entries that refer to device driver operations such as those given above which perform the necessary device dependent operations to allow communications between debug agent and monitor server.

*driver-init* : *Port*$^+$ $\xrightarrow{o}$ *Init-Result* × *Driver-Table*
*driver-init(ports)* $\triangleq$
(
  device-dependent
)

An example of the use of alternate communication channels by a monitoring system is that described by Haban [Hab90] with respect to the TMP project. The TMP monitoring system used a *separate* network though which monitoring nodes could communicate to a host debugger while the distributed application being monitored used another network over which the distributed application could communicate. This eliminated the effect of monitor network traffic over traffic generated by the distributed application.

The multiple driver approach advocated here is similar but the reasons behind its intent are different. There should be nothing to stop the PUT from using any hardware needed to implement its functionality. This includes any communication channels. The monitor cannot and should not arbitrate or monopolise the use of such resources. The approach is used primarily for simplicity, having monitor and PUT share channels fairly would increase software overhead and complexity as well as increasing the invasiveness of the monitor software.

### External Namespace—The hole through the Wall

External namespace processes messages that have the tag type, *EXTERNAL-NAME*. Messages sent to External namespace by the debug agent are processed *immediately* after they have been received and determined to have EXTERNAL-NAME tags—no messages with the EXTERNAL-NAME tag are ever enqueued. When a message is received by the Interrupt co-routine, the tag of the leading *Msg-Element* is examined. If the message tag is of type NAME then the message is enqueued to the monitor provided a session *exists*. If the message tag type is *EXTERNAL-NAME* then the message is *immediately* dispatched to External namespace to be acted upon.

The handlers within External namespace are provided to manage sessions and input-output services between debug agent and monitor. Namespace environments subordinate to External namespace provide drivers for (potentially) many devices, while the External namespace represents the device driver currently used to communicate between debug agent-monitor and (in the case of break events) between monitor and monitor.

Work-Handlers *cannot* be added or removed from External namespace. The reasoning behind this restriction is to ensure that the no malicious Work-Handlers cannot be added to circumvent session security. Further names (device drivers) may be added as children of External namespace. This behaviour is the same as that of Root namespace.

There are two session management handlers, *Open-Session* and *Close-Session* (refer to Annex E). Only *one* active session is allowed between a monitor and a debug agent for reasons given in Chapter 3. As a result, session management is quite straightforward. The only cases that have to be carefully considered are:—

a. When a debug agent attempts to open a session to a monitor that is already connected to that debug agent.

b. When a debug agent attempts to close a session that it does not "own".

types

*Open-Read-Args* ::
  *drv-name* : token;

*Open-Write-Args* ::
  *drv-name* : token
  *new-agent-addr* : *Machine-Address*;

The *new-agent-addr* argument is used to provide the monitor with the debug agent's machine address using that particular communication device. This allows replies to be sent back to the correct place.

*Close-Read-Args* ::
  *drv-name* : token;

*Close-Write-Args* ::
  *drv-name* : token;

*Send-Args* ::
  *msg* : *Message*;

*SendTo-Args* ::
  *destaddr* : *Machine-Address*
  *msg* : *Message*;

*Check-Driver-Args* ::
  *drv-name* : token
  *port* : $Port^+$;

*External-State* ::
  *connected* : **B**
  *session-src-addr* : [*Machine-Address*]
  *current-src-addr* : [*Machine-Address*]
  *read-handle* : [*Read-Proc*]
  *write-handle* : [*Write-Proc*]
  *prev-read-handle* : [*Read-Proc*]
  *prev-write-handle* : [*Write-Proc*]
  *prev-agent-addr* : [*Machine-Address*]
  *idle-op* : [*Coroutine* → *Coroutine*]
  *prev-idle-op* : [*Coroutine* → *Coroutine*];

state $\rho_{ext}$ of
  *ext-ws* : *Workspace*
  *ext-map* : *Map*
  *my-name* : *Name*
  *finder* : *Env-Handler*
  init ...; – This environment is initialised as part of Monitor global state. Refer to Annex B.
end

The *read-handle* and *write-handle* states are used to access the respective device driver routines for the currently open channel. They represent the operations that implement the actual driver operation within that driver namespace. The means by which driver functions are established and invoked is similar to abstract device driver layers provided by operating systems such as Unix and the Modula-2 language device driver interface.

The *prev-read-handle, prev-write-handle, prev-agent-addr* and *prev-idle-op* states are used by the *Switch Medium Protocol* so that roll-back can occur if the request to change communications medium fails. This protocol is discussed in detail in the next section.

The handlers of External namespace can be invoked at the time an interrupt occurs or while the Monitor co-routine is active. When an interrupt that corresponds to the receipt of a message occurs, the handler that receives a message is invoked. In the case where a message arrives at the monitor, the question to ask is what happens when another interrupt occurs while a message is being processed? If a trap occurs for some reason, normal trap processing will occur. If interrupts occur it will depend on whether the interrupt is of a greater priority than the currently processed interrupt. Since the only interrupt sources actually listened to by the monitor are communications and configuration, it is unlikely that any exclusion or serialisation problems will arise. This is because an interrupt for the same interrupt source could not occur while that interrupt was still being handled, it cannot pre-empt the currently handled interrupt because it is not of greater priority. Instead, such an occurrence might become an *interrupt pending* ensuring events remain serialised. In this way, the handlers of external namespace are *protected* from access while the interrupt is active.

External namespace also supplies handlers used for communication between target and host system (refer to Annex E). The primitive operations *Send, SendTo* and *Receive* are used for debug agent-monitor and monitor-monitor server communication. The switching of communication channels is also accomplished using *Open-Read, Open-Write, Close-Read* and *Close-Write.* It is these communication primitives within External namespace which actually interface to the driver functions that are a part of a particular device driver.

After a successful *Open-Read* all future messages are received on the channel opened. The *idle-op* operation, *stop* is used to halt the target processor. A halt state is only broken if interrupts occur. The context of the last interrupted co-routine before the interruption occurred is returned by *stop* when the halt state is broken. Trivially, the *stop* operation consists of the following:—

functions
*stop : Coroutine* $\rightarrow$ *Coroutine*
*stop(previous)* $\triangleq$
*(*
  *full-stop(previous)*
*)*

operations
*full-stop : Coroutine* $\xrightarrow{o}$ *Coroutine*
*full-stop(previous)*
*(*
  *halt();*
  return *previous*
*)*

*halt : ()* $\xrightarrow{o}$ *()*
*halt()* $\triangleq$
*(*
  skip – target-dependent
*)*

NOTE The *halt* operation could be a busy wait doing nothing productive!

*Open-Write* (refer to Annex E) returns a response to the debug agent indicating whether the handler succeeded or failed. The response generated by this handler invocation is sent to the debug agent using the newly opened channel if the handler succeeded, otherwise the result is returned

using the previous channel for writing. The information regarding the address to which to send requests to the monitor can be extracted by the debug agent from the reply message. This implies that the debug agent be actively *listening* for messages on the new channel it is switching to in addition to the old channel. This is of importance when considering the *switch medium protocol* described later in this chapter.

One of the arguments supplied to *Open-Write* is the new media address associated with that session established when the communication medium changes. This means that the *session-src-address* changes to reflect the medium through which the messages arrive. Since the address of the originator of *EXTERNAL-NAME* and *NAME* messages is checked as they arrive there is is no possibility that another debug agent can "hijack" a session subversively. Session security cannot be subverted through switching media devices.

The closing of channels is more complex due to the need to maintain state to roll-back if the switch medium protocol fails due to message loss or inability to establish connection with that new channel.

The general case regardless of the specifics of the *Close-Read* or *Close-Write* operation is similar although the state affected is differs depending upon the input-output handle involved. The procedure for closing a read or write channel is:—

(1) Determine whether roll-back can occur. If the previous handle is nil then this must be the first time a communication device has been closed. In fact closing will be impossible in this case because the open will never have been invoked in this situation.

(2) If the previous state of the handle is the same as the handler then the new handle state shall become the previous value of the current handler. Close the channel.

(3) If the previous state of the current handle was the handler then the new state of the current channel shall become the previous value of the previous handle.

## Message Reception and Transmission

The last three Work-Handlers of External namespace manage data transfer between debug agent and monitor (refer to Annex E). The *Send* handler is used to send responses to the debug agent that has a currently established session dialogue with the monitor. The *Sendto* handler is primarily used by the *Breakpoint-Handler* when the monitor has to send group breakpoint messages to other monitors. The *Receive* handler accepts messages arriving from any source, be it the debug agent or another monitor in the case of group break event messages.

All the above described handlers return an empty result body—no reply is sent to the debug agent. If they did reply then there would be effectively two responses—one for the handler reply and another for the reply associated with the success of the *Send/Receive* handler! In the case of the *Receive* handler, the value returned is the message.

operations
*Send(args : Argument, ext-ws : Workspace) res : Data* × *Handler-Status*

*Sendto(args : Argument, ext-ws : Workspace) res : Data* × *Handler-Status*

*Receive(args : Argument, ext-ws : Workspace) res : Data* × *Handler-Status*

The last handler, *Check-Driver* is an *Env-Handler* of External namespace (refer to Annex E). It is used by Device namespace to determine whether a newly inserted hardware device has an associated communication channel driver. It uses the *Port* structure determined by Device namespace

to initialise the base address of the memory map so that device drivers can initialise themselves independently of where the device actually resides in the memory map. If an associated communications device driver exists for a particular hardware device then the name of the driver is returned as part of the *Add-Hardware* handler. The debug agent can use this to switch communications media *dynamically*. The operation performed by *Check-Driver* could be considered as a specialised *Find-Name* of drivers.

operations
*Check-Driver(msg : Message,*$\rho_{ext}$ *: Env) res : Message* $\times$ *Env*

With all the external namespace handler operations defined, the auxiliaries used by the coroutines to send and receive messages now can be fully described:—

operations
*Send-Message : Message* $\xrightarrow{\circ}$ *Handler-Status*
*Send-Message(msg)* $\triangleq$
*1.* let *mk-Env(_,root-map,_,_)* = $\rho_{root}$ in
*2.*   let *mk-Env(ext-ws,ext-map,_,_)* = *root-map(EXTERNAL)* in
*3.*     let *args* = *mk-Msg-Element(⟨ARGUMENT⟩,mk-Send-Args(msg))* in
*4.*       def *mk-(_,status)* = *ext-map(SEND)(args,ext-ws)* in
*5.*         return *status*

*Sendto-Message : Machine-Address* $\times$ *Message* $\xrightarrow{\circ}$ *Handler-Status*
*Sendto-Message(to,msg)* $\triangleq$
*1.* let *mk-Env(_,root-map,_,_)* = $\rho_{root}$ in
*2.*   let *mk-Env(ext-ws,ext-map,_,_)* = *root-map(EXTERNAL)* in
*3.*     let *args* = *mk-Msg-Element(⟨ARGUMENT⟩,mk-SendTo-Args(to, msg))* in
*4.*       def *mk-(_,status)* = *ext-map(SENDTO)(args,ext-ws)* in
*5.*         return *status*

*Receive-Message : ()* $\xrightarrow{\circ}$ *Message*
*Receive-Message()* $\triangleq$
ext rd $\rho_{root}$ *: Env*
*1.* let *mk-Env(_,root-map,_,_)* = $\rho_{root}$ in
*2.*   let *mk-Env(ext-ws,ext-map,_,_)* = *root-map(EXTERNAL)* in
*3.*     def *mk-(data,_)* = *ext-map(RECEIVE)([],ext-ws)* in
*4.*       let *msg : Message* be st *msg* = *data* in
*5.*         return *msg*

### The Switch Medium Protocol

The protocol described so far has taken a simple *datagram* approach. Such an approach is desirable because it removes the need for complex state machines on the monitor—an unnecessary overhead. The debug protocol is request-reply, each message from the debug agent being self-contained. Using a datagram approach means that the monitor state used to manage protocol can be minimised. If each message is *idempotent*, reliability is further enhanced. If a message time-out occurs, the debug agent can resend the message without needing to be concerned about (possible) cumulative effects.

For example, writing to memory at a given block of locations is an idempotent operation, it can be retried any number of times, each retry behaving identically. The only connection state maintained by the monitor is to which debug agent the monitor is connected and whether a connection currently exists. The only caveat in this monitor and protocol model is that the *monitor* cannot time-out. Messages sent by the monitor to other monitors are sent without expectation of reply, this means that external breakpoints notification messages may never be received by the other monitors. For effective debugging to take place however, the medium must in any case be

*reliable.* If the protocol was designed to handle unreliable media, then the monitor would become more complex possibly affecting the PUT's execution in the process.

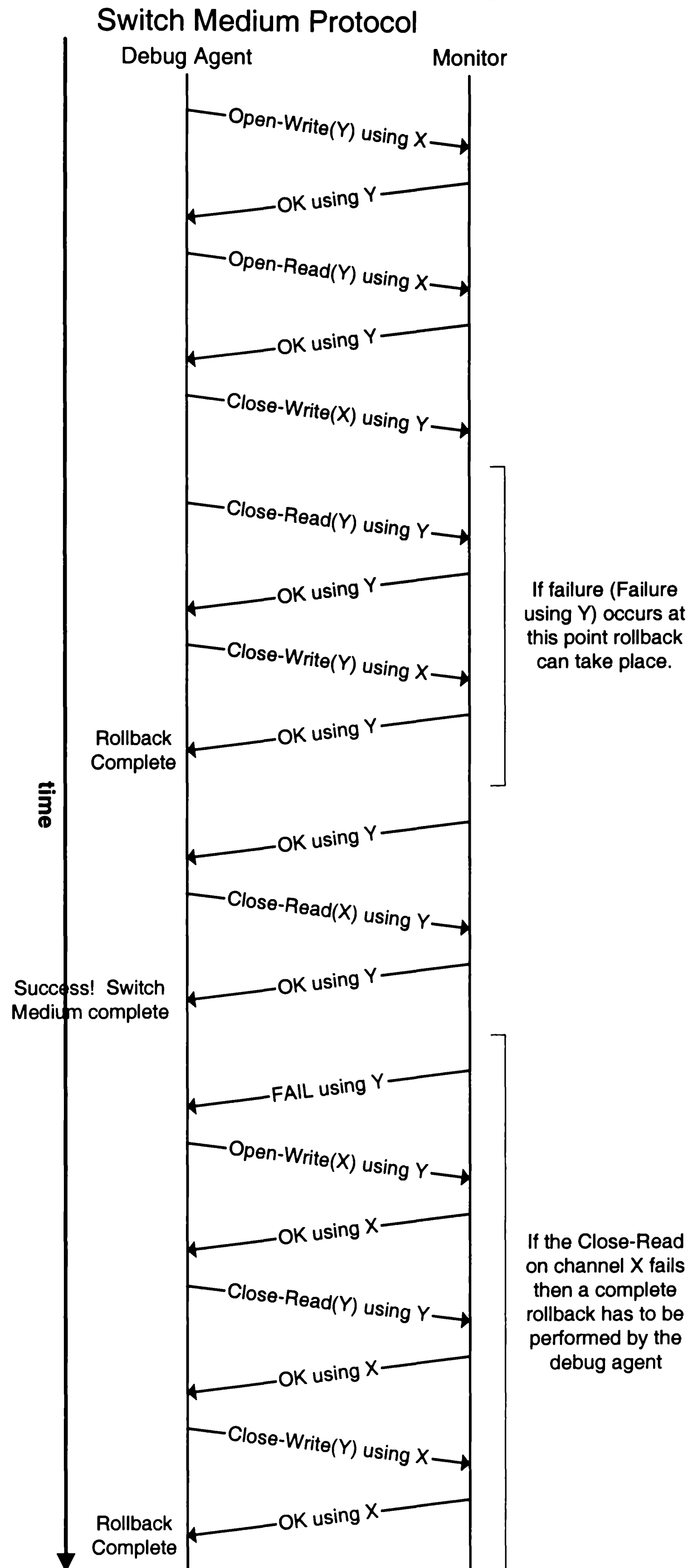



Fig. 7.3 *Agent protocol for switching communication channels*

If protocol is broken down to its most basic operations, the debug agent can compose these primitives and implement higher order behaviour in any manner it likes. Barring the need to

establish a session and switch communication channels which are required protocol state machines of any debug agent implementation, all other protocol interactions depend on how the debug agent decides to interpret responses to requests.

The switching of communications channels uses External namespace primitives. The state machine establishes the new input and output channels before closing the old input and output channels. Message control flows are given in the form of an *Action using Channel* in Fig 7.3 where the action is the protocol message and the channel is that used to send the request or reply over.

There are two roll-back stages which may take place to account for the failure of *Close-Read* or *Close-Write*. The first two phases of *Open* are trivial to roll-back. The first *Open-Write* need not do anything if failure occurs, the second *Open-Read* need only re-establish the previous write handle by using *Open-Write* on the previous device.

The actual protocol for switching channels is simple if no message loss or failure to open or close channels occurs. Given a hypothetical situation where device X provides the current channel and device Y is to be switched to there are only four steps that take place:—

(1) *Open-Write(Y)* using channel X as the message carrier.

(2) *Open-Read(Y)* using channel X as the message carrier. By the end of this phase, channel Y will be available for use in both input and output. Roll back state will have been established by the end of this phase.

(3) *Close-Write(X)* using channel Y. Shutdown the output channel.

(4) *Close-Read(X)* using channel Y. Shutdown the input channel. The channel switching is now complete and normal message processing can continue.

By the end of the second phase, messages sent using channel X will effectively be ignored as the handles for channel Y will have been established and the monitor will be only actively listening and receiving messages sent along channel Y in the future.

### Summary

The contents of this chapter considered the infrastructure required to establish communications links between debug agent and monitor so that messages could be sent and received by the monitor.

It is the responsibility of *External* namespace to manage a channel of communication. Subordinate to External namespace are *Communications* namespace(s). Each communication namespace contains driver primitives for opening, closing, reading and writing. External namespace represents the current active driver and the current session state. Because the PUT may potentially use existing channels of communication media, the monitor provides a mechanism where the channel used by the monitor to communicate with the debug agent can be switched as required.

# Chapter 8

# Operation Decomposition

The purpose of this chapter is to derive explicit forms of specification for the key operations and functions in the monitor.

This chapter gives examples of the operational decomposition process that must be carried out to derive a concrete implementation from the largely implicit specifications given so far. The topic of the debug agent and target monitor protocol message reification is discussed in the following chapter.

It is the general applicability of the model that is being established by this thesis. Operational decomposition in this sense is *not* an implementation as such. It is a *template* or *rubber stamp* mechanism that with appropriate tool support can be used to produce a number of refined implementations that will work together between themselves and a debug agent provided each satisfies the specification and message protocol.

It will be remembered up to this point that various specifications have been noted as being *target-dependent.* The term target-dependent implies:—

a. The means of achieving the aim may be different for different CPUs but the goal itself ultimately must satisfy the specification.

b. *Different* hardware specific techniques may be used to implement required functionality that satisfies the model specification.

c. The issue of *device-dependence* needs to be taken into account. Device dependent operations rely upon the facilities peculiar to a device to implement required behaviour. The implementation of the behaviour is only required to satisfy the post-condition specified, how this may be achieved is irrelevant.

d. There is no point in further developing these target-dependent specifications as the implementation step immediately constrains the implementer to a given target configuration.

e. There is a need for a handful of assembly language instructions.

### Target Independent Decomposition

Every eventual implementation depends upon the style and wishes of the implementer up to the point where it has to satisfy the specification "contract". The various possible refinements of the specification will eventually yield these particular implementations.

Most of the handlers and co-routine event loops presented so far are target-independent. Only a fraction (maybe one percent) of the code requires an implementation specific to a particular hardware device or CPU. For example, namespace model constructors and destructors are target-independent and will have implementations that are identical irrespective of the target environment.

Target dependencies are likely to affect the following sections of a monitor implementation:—

a. Context saving and restoration operations. These were the *lambda* expressions specified as part of the monitor low level operations.

b. Manipulation of raw context to store and retrieve contexts and workspace areas.

c. Code that manipulates a hardware resource, target or device specific code will require specific registers or addresses to be written or read.

Target dependencies that arise as a result of these situations appear in some of the model's functions and operations. However, the explicit derivations presented here have been built in such a way that 99% or more of the code necessary to implement the monitor is free of target dependencies.

Given that target dependencies are well isolated by the model, practically all operations and functions can be generated by tools that build program source text from the explicit derivation. The issue of generating implementations for several (different) targets then becomes that of implementing the above behaviour—a necessary, somewhat mundane step.

## Notes on the Derivations Presented

To prevent the argument from becoming too drawn out because of a great number of explicit derivations, only important derivations are discussed here, the remainder being placed with appropriate explanation in the body of Annex D.

This chapter concentrates therefore on explicitly deriving operational decompositions for the following:—

(1) The model constructors and destructors, Find-Name, Add-Name along with the "specialised" entity handlers of Target namespace.

(2) The handlers within Target namespace used to establish, modify and clear breaks with particular attention being paid to the breakpoint handling algorithm.

(3) The initialisation required to bootstrap the monitor server and hence the namespace model.

The argument length checking component for each of the handlers specified is *not* required in the following derivations. The existence and type checking of argument packs for *model* handlers is performed by *retrieval* functions. Message *retrieval* and *concretion* functions will be defined in the following chapter. Message correctness is checked by retrieval functions before the messages are enqueued or acted upon. In this way, any message processed by the namespace engine within the monitor may reasonably assume messages are in the correct format.

For standard operational decomposition rules, see Annex I.

## Operational Decomposition

The goal of operational decomposition is to transform *implicit specifications* describing the model into an *executable program* such that the explicit derivation establishes the specification's *post-condition.* This transformational approach is reinforced with proof obligations to satisfy the requirement. For example, do these statements in the explicit derivation satisfy the assertions of the specification? Decompositions yield "executable" VDM programs, the program text being composed of operation sequences, choices and iteration.

The derivations presented here are *operational decompositions*, code structures are derived to an extent where tools such as the IFAD VDM Toolbox [IFAD99] can be used to interpret the VDM or generate code to be compiled.

The derivations, however, do not consider *data refinement*—the actual concretions of model data structures. These (with the exception of the wire protocol) are left to the implementer. Less than 1% of the data structure refinement (context save areas, interrupt stack frames and parts of device drivers) depends on hardware constraints or efficiency requirements.

There are many avenues by which data type refinement can take place. For example, VDM-SL map types may be implemented by the use of arrays, composite types may use records, sequences may use linked lists or arrays of a certain type and optional types may be implemented by the use of a *variant* record. The use of the abstract *set* type can, in the simple case of bit sets be easily refined. The sets of arbitrary types that can be used in a *VDM-SL* specification represent greater difficulty—a refined implementation may use an array data structure to contain the set data elements. The various set operators would then have to be defined over this representation as refined for that implementation.

Data refinement decisions are motivated by the demands of efficiency or possibly even by ease of implementation! A stack abstraction could be represented as some finitely-bounded array or a more flexible linked list structure. Each has its own advantages and trade-offs—something that an implementer must consider when making a choice. As long as a concretion of that model type is *adequate* and preserves the meaning implied by the specification, the question of *how* it is implemented does not matter.

It is important to consider that while specifications and various program refinement "rules" allow the construction of program code there is still a considerable "inspiration" factor involved; there may be many ways by which a program could be derived from a specification. Take for example, the case of the multiplication of two integers. Two solutions come immediately to mind, repeated addition and the shift-add method. A program that uses either of these algorithms is correct provided the post-condition is satisfied. Some very abstract specifications may concisely communicate the end result while leaving considerable freedom for the implementer to express algorithms to produce that result.

Most of the specifications given in this thesis are not of this nature, but do give rise to program code whose derivation roughly mimics that of the specification. There is, however, considerable flexibility with respect to *data refinement* issues. The ability to reason at a more abstract level is therefore useful when considering the properties of a model without getting mired in unnecessary detail. When implementation is undertaken, the data refinement phase ensures that the mapping from model abstract type to concrete representation is carried out correctly.

### Referencing Namespace Workspace

Both *Env-Handlers* and *Work-Handlers* need to access and manipulate environment state. Namespace state is different to monitor global state. Each namespace environment has its own *workspace* area—this state can only be accessed by handlers of that namespace environment. While each environment is separate, the links between namespace environments form a tree of namespace environments whose root is part of the Monitor global state.

The auxiliary operations described in Chapter 6 operate on monitor global state performing operations on the message queue, co-routine stack and trap state. Monitor state is statically determinable and pervasive; such state does not disappear, it persists throughout the monitor's lifetime, inside and outside a session between debug agent and monitor. Namespace state is potentially *transient*. With the exception of the mandatory monitor namespaces such as Target and External namespace, the namespace object tree is subject to changes resulting from the addition and removal of namespace environments. For example, during the testing of a user program, entities

subordinate to target namespace may exist for the lifetime of the PUT only to disappear when the PUT runs to completion. Each entity environment's state is valid *only* when the PUT is executing and that entity exists as a child namespace of Target namespace.

Namespace state cannot be referred to in the same manner as monitor global state. The namespace state is *relative* to its environment and the handlers of that environment which manipulates that state. An *Env-Handler* or *Work-Handler* has to:—

(1) Unbind the state component of the Workspace for the environment.

(2) Declare a state component whose name is synonymous with the real environment state.

(3) Modify or access the values in the namespace state as required.

The prologue to any handler that uses and/or modifies the environment state accomplishes steps (1) and (2) above.

```
Example-Work-Handler : Argument × Workspace -o-> Data × Handler-Status
Example-Work-Handler(args,ws) ≜
(
  let mk-Workspace(_,_,old-state) = ws in
  (
    dcl new : X-State = old-state;
  )
)
```

The handler body accomplishes step (3).

```
(
  let mk-Workspace(_,_,old-state) = ws in
  (
    dcl new : X-State = old-state;
    new.s1 := ...;
    ⋮
    new.sn := ...; – Workspace gets modified values.
  )
)
```

## Namespace Model Derivations

From an operational standpoint the model handlers are reasonably straightforward to derive explicitly. The most difficult to derive is the *Remove-Name* handler which requires the recursive traversal and subsequent removal of the descendent environments. The key observation is:—

> *All the model handlers act upon the environment to which they are bound. Each model handler modifies the map and/or the workspace of that environment.*

This observation implies closure within the current environment for model handlers. It will be remembered that the namespace model is closed with respect to:—

(1) Handler access via *Find-Name*.

(2) *Work-Handlers* only affect the workspace of the namespace to which that handler is bound.

(3) The addition and removal of namespaces.

### Find-Name

*Find-Name* is an inherently recursive definition whose derivation is reasonably straightforward. For the sake of extra clarity, the auxiliary operation, *dispatch-name* is used.

```
Find-Name : Message × Env → Message × Env
Find-Name(msg,ρ) ≜
1. (
2.    let nm = hd msg,rest = tl msg in
3.    (
4.      if Message-Type(nm) ≠ ⟨NAME⟩ then
5.        return mk-Handler-Result([mk-Msg-Element(⟨ERROR⟩,EXPECTED-NAME)],ρ)
6.      else
7.        return dispatch-name(nm,rest,ρ)
8.    )
9. )
```

The *dispatch-name* operation resolves names within a given environment. If the name at the head of the message is not found then a name resolution error is returned to the debug agent. This can be used by the debug agent to retry the message with the correct pathname.

If the name exists in the environment, the operation performed depends on the type of object mapped to by the name and whether there is any remaining message body—possibly an argument pack or another name.

### Add-Name

The *Add-Name* operation constructs an environment whose immediate ancestor is the current environment. The map of the current environment is modified to reflect that a new subordinate environment is "connected" to the current environment.

In a fully refined implementation of the model the map will probably be some finite structure. This will probably be an array of procedures with a tag representing whether the mapped value is an *Env*, *Env-Handler* or *Work-Handler*. Being a finite map, the implementation must consider the issues where a name may not be able to be added, not because of a lack of workspace but because possibly there is not enough space to create an entry on the map.

```
Add-Name : Message × Env -°→ Message × Env
Add-Name(msg,ρ) ≜
1. (
2.    let mk-Env(ws,loc-map,my-name,finder) = ρ in
3.    (
4.      if Message-Type(msg(1)) = ⟨ARGUMENT⟩ then
5.      (
6.        let mk-Add-Name-Args(name,size) = Message-Value(msg(1)) in
7.        (
8.          if name ∉ dom loc-map then
9.          (
10.           let mk-Env(_,d-map,_,d-finder) = ρ_default in
11.             def space = Acquire-Space(size) in
12.             (
13.               – Create child default environment
14.               – Add child environment to the map
15.             )
16.         )
17.         else
18.         (
19.           return mk-([mk-Msg-Element(⟨ERROR⟩,NAME-ALREADY-EXISTS)], ρ)
```

```
20.             )
21.           )
22.       )
23.       else
24.       (
25.          return mk-([mk-Msg-Element(⟨ERROR⟩,ARGUMENT-EXPECTED)], ρ)
26.       )
27.   )
```

### Remove-Name

The removal of namespace environments is more complex because of the need to remove subordinate environments that may be nested deeper in the hierarchy. Unlike the other namespace constructors and destructors presented in this section, *Remove-Name* requires more effort to satisfy the requirements original specification presented in Chapter 4.

```
Remove-Name : Message × Env ⥲ Message × Env
Remove-Name(msg,ρ)  ≜
1.  (
2.    let mk-Env(_,loc-map,_,_) = ρ in
3.      if Message-Type(msg(1)) = ⟨ARGUMENT⟩ then
4.        let mk-Remove-Name-Args(name) = Message-Value(msg(1)) in
5.          if name ∈ dom loc-map then
6.            if is_Env(loc-map(name)) then
7.              return do-child-remove(loc-map(name),ρ)
8.            else
9.              return mk-([mk-Msg-Element(⟨ERROR⟩,NOT-AN-ENV)],ρ)
10.         else
11.           return mk-([mk-Msg-Element(⟨ERROR⟩,NAME-NOT-FOUND)],ρ)
12.     else
13.       return mk-([mk-Msg-Element(⟨ERROR⟩,ARGUMENT-EXPECTED)],ρ)
14. )
```

The auxiliary operation, *do-child-remove* works by entering the child environment to be removed and invoking *Remove-Name* in that child environment for any children of the child environment. When all children of the child environment are removed, that child environment is removed and the name maplet is deleted from the parent environment. The operation is recursive with the key being that a *Remove-Name* is invoked for each subordinate environment. A loop is used to iterate over all names in the map of each child namespace so that all subordinate environments are removed.

### Add-Handler

The last three model constructor and destructor handlers (*Add-Handler*, *Remove-Handler* and *Replace-Handler*) are characterised by their use of workspace management operations to load and unload handlers within a given environment. Because the current environment *must* exist for these handlers to be invoked they cannot affect any workspace that is not their own. This means that these latter three handlers are "closed" in the sense that their "maliciousness" is limited only to what is theirs. This security is important as these handlers deal only with *Work-Handlers*. *Env-Handlers* are considerably more powerful because they can access the map. If the debug agent were allowed to add, remove, or replace *Env-Handlers*, the security of the namespace model could be seriously compromised. Monitor handlers access the map and subordinate namespace state only in circumstances where the interaction is well controlled and regular. In most cases these are namespaces that were originally created by messages that originate from events such as dynamic configuration or registration of entities.

Unlike the addition of a namespace, *Add-Handler* is only required to manage the workspace of the current environment to "load" the handler into workspace.

```
Add-Handler : Message × Env -o-> Message × Env
Add-Handler(msg,ρ) ≜
1.  (
2.     let mk-Env(ws,loc-map,my-name,finder) = ρ in
3.       if Message-Type(msg(1)) = ⟨ARGUMENT⟩ then
4.       (
5.          let mk-Add-Handler-Args(name,size,body) = Message-Value(msg(1)) in
6.            if name ∉ dom loc-map then
7.            (
8.               – attempt to copy the handler into the free workspace
9.               – if the copy is successful then map the name
10.               – else return an error
11.            )
12.            else
13.            (
14.               return mk-([mk-Msg-Element(⟨ERROR⟩,NAME-ALREADY-EXISTS)],ρ)
15.            )
16.       )
17.       else
18.       (
19.          return mk-([mk-Msg-Element(⟨ERROR⟩,ARGUMENT-EXPECTED)],ρ)
20.       )
21.  )
```

## Remove-Handler

The *Remove-Handler* operation presents none of the problems in the derivation of the *Remove-Name* handler. The reason behind this is simple—removing a solitary handler requires only the manipulation of the current environment's map and workspace (where the handler is stored).

```
Remove-Handler : Message × Env -o-> Message × Env
Remove-Handler(msg,ρ) ≜
1.  (
2.     let mk-Env(ws,loc-map,my-name,finder) = ρ in
3.       if Message-Type(msg(1)) = ⟨ARGUMENT⟩ then
4.       (
5.          let mk-Remove-Handler-Args(name) = Message-Value(msg) in
6.            if name ∈ dom loc-map then
7.            (
8.               if is_Work-Handler(loc-map(name)) then
9.               (
10.                 – free the space occupied by the handler
11.                 – if the space has been freed unmap the name
12.                 – else return an error
13.               )
14.               else
15.               (
16.                 return mk-([mk-Msg-Element(⟨ERROR⟩,NOT-A-WORK-HANDLER)],ρ)
17.               )
18.            )
19.            else
20.            (
21.               return mk-([mk-Msg-Element(⟨ERROR⟩,NAME-NOT-FOUND)],ρ)
22.            )
23.       )
```

```
24.      else
25.      (
26.         return mk-([mk-Msg-Element(⟨ERROR⟩,ARGUMENT-EXPECTED)],ρ)
27.      )
28. )
```

### Replace-Handler

Besides the *Remove-Name* handler, the *Replace-Handler* operation is probably the next most complex of the model handlers presented. There are two methods by which the goal could be achieved—a replace operation can be modelled as a remove handler of a given name followed by an add of the same name (with a different handler) or it can be treated as an inline replace where the name is *never* unmapped. The latter view has been taken in this case.

```
Replace-Handler : Message × Env ─o→ Message × Env
Replace-Handler(msg,ρ) ≜
1. (
2.    let mk-Env(ws,loc-map,my-name,finder) = ρ in
3.      if Message-Type(msg(1)) = ⟨ARGUMENT⟩ then
4.      (
5.         let mk-Replace-Handler-Args(name,size,body) = Message-Value(msg(1)) in
6.           if name ∈ dom loc-map then
7.           (
8.              if is_Work-Handler(loc-map(name)) then
9.              (
10.                – determine whether the new handler can be copied into free workspace
11.                – if the copy was successful then remove the old handler
12.                – remap the name if the old handler was successfully removed
13.                – otherwise return an error
14.             )
15.             else
16.             (
17.                return mk-([mk-Msg-Element(⟨ERROR⟩,NOT-A-WORK-HANDLER)],ρ)
18.             )
19.          )
20.          else
21.          (
22.             return mk-([mk-Msg-Element(⟨ERROR⟩,NAME-NOT-FOUND)],ρ)
23.          )
24.      )
25.      else
26.      (
27.         return mk-([mk-Msg-Element(⟨ERROR⟩,ARGUMENT-EXPECTED)],ρ)
28.      )
29. )
```

## Special Env-Handlers

The following *Env-Handlers* are special in the sense that they are not necessarily invoked by the debug agent. Each represents a specialisation of *Add-Name* or *Remove-Name* functionality. These handlers are invoked by local events—the addition of new hardware, the addition or removal of entities. Unlike the namespace constructor/destructors their response values are not of type RESPONSE but rather they are of type RESPONSE-DATA. This is due to the nature of invocation—a debug agent does not typically send messages that invoke the following services:—

a. *Add-Hardware*—this handler is normally invoked in response to dynamic configuration events

b. *Add-Entity*—this handler is invoked in response to entities being registered (possibly at the start up of the PUT).

c. *Remove-Entity*—this handler is invoked when entities are being deregistered by the PUT or possibly on shutdown or when the object represented by the entity finalises.

Specialised forms of *Find-Name* also exist. These perform operations whose behaviour is similar to that of *Find-Name* but carry out additional extra functions:—

a. *Find-Entity*—this *Env-Handler* is part of Target namespace and determines the entity active at the time a trap occurred by using workspace information. This is accomplished by doing a bounds check on each entity registered.

b. *Check-Driver*—this is part of External namespace and is used to determine whether a hardware device has an associated device driver that can be initialised for use by the monitor. The *Check-Driver* handler uses port address information passed to it to initalise the driver registers.

While each of these handlers is specialised for a given namespace environment the actual operational effect can be broken down into two components.

a. Create (or possibly destroy) the environment. This might mean establishing a child namespace environment or removing that environment.

b. Manipulate or inspect the state of that child environment.

Each of the handlers referred to in the previous paragraphs exhibits this pattern of construction.

The handlers of Target namespace provide three variations of specialised *Env-Handlers* that manipulate the *map* to support program entities:—

(1) *Find-Entity* is a find operation with a difference, instead of resolving a handler given a name as in a typical *Find-Name* operation, a name of the entity is resolved using locality information to match attributes of that entity.

(2) *Remove-Entity* is used to remove both the entity's namespace but also to manipulate the break table in such a manner that all references to that entity are also removed from the break table.

(3) *Add-Entity* adds new environments to Target namespace and then establishes entity state used for determining the identity of the entity that was running at the time an event occurred.

### Find-Entity

The *Find-Entity* handler uses the *do-find-entity* auxiliary operation to determine which entity was executing when a trap occurred.

```
operations
Find-Entity : Message × Env ⥲ Message × Env
Find-Entity(msg,ρtgt) ≜
1. (
2.    let mk-Find-Entity-Args(t-msg) = Message-Value(msg(1)) in
3.      let mk-Env(_,tgt-map,_,_) = ρtgt in
4.        let mk-Trap-Msg(ctxt,_) = t-msg in
5.          let mk-Context(what,_,_,_) = ctxt in
6.            return mk-(do-find-entity(tgt-map,what,ρtgt),ρtgt)
7. )
```

```
do-find-entity : Map × Address × Env -o-> Message
do-find-entity(map,what,ρ) ≜
1.  (
2.      – for all entities in the target namespace map
3.      – if a trap context lies within an entity workspace return the entity name
4.      – otherwise it could be PUT or monitor itself
5.  )
```

Both *Find-Name* and *Find-Entity* are similar in that *name resolution* takes place. However, the parameters that both handlers take are quite different and as a result the end action taken is markedly different.

a. *Find-Name* resolves the *supplied* name to another environment, *Env-Handler* or *Work-Handler*. In the latter two cases, direct action is taken—the execution of a handler. In the former case, *Find-Name* is invoked within the child environment.

b. *Find-Entity* determines the name of the entity by testing whether each entity "encloses" the interrupted context. The result is that a name is returned, no action is performed as a result of the reverse name resolution that took place.

The original specification of Find-Entity required a particular search order. The order stated was that the PUT entity was the last *program* entity to be tested against. It can be seen that the above *do-find-entity* conforms to this specification. The search map is constrained only to those entities in the Target namespace map. The search sequence is ensured by the fall-through case—if the search gets past the loop then the entity interrupted can only either be the PUT or the monitor itself (which, it will be remembered, *can* cause a break or trap event because of the ability to define user *Work-Handlers*).

### Add-Entity

Of the three handlers presented, *Add-Entity* is probably the most like *Add-Name*. Adding an entity is just adding a namespace to Target namespace and establishing the entity's state to reflect that entity's workspace area.

```
operations
Add-Entity : Message × Env -o-> Message × Env
Add-Entity(msg,ρtgt) ≜
1.  (
2.      if Message-Type(msg(1)) = ⟨ARGUMENT⟩ then
3.      (
4.          let mk-Add-Entity-Args(e-name,base,size,pc-ptr,final-ptr) = Message-Value(msg) in
5.              let mk-Env(_,map,my-name,_) = ρtgt in
6.                  def [ent-res,_] = Add-Name(mk-Msg-Element(⟨ARGUMENT⟩,
7.                                      mk-Add-Name-Args(e-name,default-ws-size)),ρtgt) in
8.                  (
9.                      if Message-Type(ent-res) ≠ ⟨RESPONSE-DATA⟩ then
10.                     (
11.                         return mk-(ent-res,ρtgt)
12.                     )
13.                     else
14.                     (
15.                         – initialise entity state
16.                         – return entity name
17.                     )
18.                 )
19.     )
```

```
20.     else
21.     (
22.         return mk-([mk-Msg-Element(⟨ERROR⟩,ARGUMENT-EXPECTED)],rhotgt)
23.     )
24. )
```

### Remove-Entity

The *Remove-Entity* operation not only has to remove the entity namespace but also any references to the entity within a given break entry. The key point to remember is that within a set of break entries, the entity sets of each entry are *disjoint* as per the *Break-Entry* invariant. This means that an entity can occur at most *once* within that entry set (or not at all). *Remove-Entity* removes the entity name from the entry that has that entity as a group member.

```
operations
Remove-Entity : Message × Env -o-> Message × Env
Remove-Entity(msg,ρtgt) ≜
1.  (
2.     let mk-Env(tgt-ws,_,my-name,_) = ρtgt in
3.        let mk-Workspace(_,_,tgt-state) = tgt-ws in
4.           let mk-Target-State(_,_,_,_,_,_,breaks,code-map,_) = tgt-state in
5.              let mk-Remove-Entity-Args(entity) = Message-Value(msg) in
6.              (
7.                 remove-entity-break-entries(entity,tgt-ws);
8.                 def [removed,_] = Remove-Name(msg,ρtgt) in
9.                    if Message-Type(removed(1)) = ⟨RESPONSE⟩ then
10.                      return mk-([mk-Msg-Element(⟨RESPONSE-DATA⟩,
11.                                              [my-name,entity])],ρtgt)
12.                   else
13.                      return mk-(removed,ρtgt)
14.             )
15. )
```

The *remove-entity-break-entries* auxiliary operation removes any reference to the entity being removed from all break entries of which it is a member.

```
operations
remove-entity-break-entries : Entity × Workspace -o-> ()
```

## The Break Handlers of Target Namespace

Most of the monitor server's complexity lies in the establishment and handling of break events when they occur.

The entities model provides a means of establishing a less invasive *task-relative* break event mechanism. The PUT can only be halted if there is a break event set on that entity. Other entities for which a break event has not been established can continue their execution with minimal effects from the intrusion necessary to determine whether the entity previously executing was "under the influence" of that break event.

It will be remembered that there are three types of break event:—

(1) Local break events

(2) Group break events

(3) Global break events

The most complex events to handle are group break events. This complexity is due to the issues of having to potentially establish *delayed* break events. It will be remembered that a group break event specification can span multiple targets. The delayed break event is used to halt entities that are members of the same group break event notification that arrived from some other target. When a group break event occurs, its occurrence is signalled to all members of the group. A delayed break event is set on the entities that *should* be halted by that group break event occurrence if the interrupted entity was not a member of that group break specification—when these entities are later resumed (as the result of whatever scheduling method used by the PUT) they will be immediately halted, thus satisfying the group break. If it was a member then the halt should occur immediately.

Somewhat less complex is the handling of global break events. Global break events cause an immediate cessation of activity on that target (and other targets) with each registered entity executing its finalisation code before terminating the PUT.

Local break events are the least complex but cause the greatest number of possible outcomes. A break event occurring locally could be a local, group or global break occurrence. In the two latter situations the occurrence of either event may have an effect on other target systems.

### The Break-Handler

The *Break-Handler* is invoked as the result of local break event occurrences, group break event messages from other monitors or global break event messages from the debug agent. Invocation of the *Break-Handler* is different in that it is acted upon as the event occurs, it is not enqueued in the way that other debug agent *requests* are handled.

```
operations
Break-Handler : Argument × Workspace -o-> Data × Handler-Status
Break-Handler(args,tgt-ws) ≜
1.  (
2.     let mk-Workspace(_,_,old-tgt-state) = tgt-ws in
3.        let mk-Break-Handler-Args(brk-msg,ρ) = Message-Value(args) in
4.           let mk-Breakpt-Msg(trapmsg,_) = brk-msg in
5.              let mk-Trap-Msg(_,kind) = trapmsg in
6.                 def mk-(halting,reply) =
7.                 cases kind:
8.                 ⟨LOCAL-BREAK⟩ → local-break-handler(brk-msg,ρ),
9.                 ⟨GROUP-BREAK⟩ → group-break-handler(brk-msg,ρ),
10.                 ⟨GLOBAL-BREAK⟩ → global-break-handler(brk-msg,ρ)
11.                end in
12.                (
13.                   dcl new-tgt-state : Target-State := old-tgt-state;
14.                   if halting then
15.                   (
16.                      if new-tgt-state.run-state ∈ {⟨RUNNING⟩,⟨TERMINATING⟩} then
17.                      (
18.                         run-state := ⟨STOPPED⟩;
19.                      )
20.                      else
21.                      (
22.                         skip;
23.                      )
24.                      return mk-(reply,⟨SUCCESS⟩)
25.                   )
26.                   else
27.                   (
```

```
28.                 def interrupted = Pop-CRStack() in
29.                 (
30.                    if interrupted = ⟨MONITOR⟩ then
31.                    (
32.                       return mk-([CONTINUING],⟨SUCCESS⟩)
33.                    )
34.                    else
35.                    (
36.                       if new-tgt-state.run-state = ⟨TERMINATING⟩ then
37.                       (
38.                          push-continue();
39.                          return mk-(reply,⟨SUCCESS⟩)
40.                       )
41.                       else
42.                       (
43.                          push-continue();
44.                          return mk-([CONTINUING],⟨SUCCESS⟩)
45.                       )
46.                    )
47.                 )
48.              )
49.           )
50. )
```

If the break handler auxiliary operations determine that the PUT should be halted, the run-state is set to stopped if the PUT was previously running or finalising. If break processing determines that the PUT should not be halted a message is enqueued at the front to restart the PUT after the reply has been sent to the debug agent.

## Setting, Modifying and Clearing Breaks

The handler operations that establish break events are the most complex of Target namespace. *Set-Break, Modify-Break* and *Clear-Break* to a lesser degree lie at the core of the break event model. The group break and delayed break handling relies upon the ability to establish break events dynamically as the PUT executes. The *Remove-Entity* handler also manipulates the break entries so that break events are completely removed when all the entities within an entry set have been removed.

The *Set-Break* handler establishes a *new* break entry at a given address. If a break event with that break identity already exists then the operation fails.

```
operations
Set-Break : Argument × Workspace -o-> Data × Handler-Status
Set-Break(args,tgt-ws) ≜
1. (
2.     let mk-Workspace(_,_,tgt-state) = tgt-ws in
3.        let mk-Target-State(_,_,_,_,_,_,_,breaks,_,_) = tgt-state in
4.           let mk-Set-Break-Args(kind,name,_,entity) = Message-Value(args) in
5.              let id = mk-Break-Identity(kind,name) in
6.              (
7.                 if id ∈ dom breaks then
8.                    return mk-([BREAK-ID-ALREADY-EXISTS],⟨FAILURE⟩)
9.                 else
10.                 (
11.                    def result = set-entry(args,tgt-ws) in
12.                    (
13.                       if result = [BREAK-SET-OKAY] then
14.                       (
```

```
15.                         push-add-predicate(entity,id);
16.                         return mk-(result,⟨SUCCESS⟩);
17.                     )
18.                     else
19.                         return mk-([BREAK-SET-FAILED],⟨FAILURE⟩);
20.                 )
21.             )
22.         )
23. )
```

The *Modify-Break* handler modifies an *established* break entry.

```
operations
Modify-Break : Argument × Workspace ⥲ Data × Handler-Status
Modify-Break(args,tgt-ws)  ≜
1. (
2.     let mk-Workspace(_,_,tgt-state) = tgt-ws in
3.         let mk-Target-State(_,_,_,_,_,_,breaks,_,_) = tgt-state in
4.             let mk-Modify-Break-Args(kind,name,old-pc,_,_,
5.                     _,new-entity,_) = Message-Value(args) in
6.                 let id = mk-Break-Identity(kind,name) in
7.                 (
8.                     if id ∈ dom breaks then
9.                     (
10.                        – determine whether entry exists
11.                        – if it does then modify it,
12.                        – otherwise add it
13.                    )
14.                    else
15.                    (
16.                    return mk-([CANNOT-MODIFY-NON-EXISTENT-BREAK],⟨FAILURE⟩);
17.                    )
18.                )
19. )
```

*Clear-Break* takes the breakpoint identity and the address and removes the breakpoint from the break map and code map.

```
operations
Clear-Break : Argument × Workspace ⥲ Data × Handler-Status
Clear-Break(args,tgt-ws)  ≜
1. (
2.     let mk-Workspace(_,_,tgt-state) = tgt-ws in
3.         let mk-Target-State(_,_,_,_,_,_,breaks,_,_) = tgt-state,
4.               mk-Clear-Break-Args(id,addr) = Message-Value(args) in
5.         (
6.             if id ∈ dom breaks then
7.             (
8.                 def b-set = breaks(id) in
9.                     def exists = entry-with-address(b-set,addr) in
10.                        if exists then
11.                        (
12.                            return mk-(clear-entry(args,tgt-ws),⟨SUCCESS⟩);
13.                        )
14.                        else
15.                        (
16.                            return mk-([BREAK-ENTRY-NOT-FOUND],⟨FAILURE⟩);
17.                        )
18.            )
19.            else
```

```
20.         (
21.             return mk-([BREAK-ID-NOT-FOUND],⟨FAILURE⟩);
22.         )
23.     )
24. )
```

The *clear-entry* auxiliary removes all entity predicates for all entities associated with a break event and then removes the break event from the break table.

## Controlling Program Execution

Loading and testing the PUT consists of two phases; loading the code/data and the establishment of the PUT context so that the PUT can be transferred to at some future instant.

The *Load Protocol* is a combination of these two phases. The first phase, the loading of code and data, is accomplished by sending messages to Memory namespace. The *Initialise-Program* handler is used to establish the PUT context. The *Continue-Program* handler is used to commence the execution of the PUT.

Initialisation (lines 19-27) ensures that any previously registered entities from another test run are removed in addition to any break table entries. If these were not removed, the possibility of interference could exist due to stale information from previous program test runs. Because break events are entity *relative*, the use of *Remove-Entity* ensures that the break table is also cleared—if the entity set for a given break entry is empty then that break event is removed.

The operation call, *set-termination* is used to establish the wind-up phase of the PUT. In a case where normal termination occurs, this effectively means the transfer of control to the monitor co-routine along with a "reason" denoting the successful termination of the PUT.

The *run-state* is set to LOADED, indicating that the PUT context is initialised so that transfer of control can later take place.

```
operations
Initialise-Program : Message × Env -o-> Message × Env
Initialise-Program(msg,ρtgt) ≜
1. (
2.     let mk-Env(tgt-ws,tgt-map,-,-) = ρtgt in
3.       let mk-Workspace(-,-,old-tgt-state) = tgt-ws in
4.         let mk-Initialise-Program-Args(init-pc,init-sp,put-base,put-size) =
5.             Message-Value(msg(1)) in
6.           (
7.             dcl new-tgt-state : Target-State := old-tgt-state;
8.             new-tgt-state.put-ws-base := put-base;
9.             new-tgt-state.put-ws-size := put-size;
10.             new-tgt-state.put-pc-ptr := put-pc-dump-base;
11.             new-tgt-state.put-finalisation := nil;
12.             new-tgt-state.saved-id := nil;
13.             new-tgt-state.program := ⟨PROGRAM-UNDER-TEST⟩;
14.             new-tgt-state.program-under-test :=
15.               create-coroutine(init-pc,init-sp,
16.                                put-reg-dump-base,
17.                                new-tgt-state.program,
18.                                Program-Under-Test);
19.             def entities = map-to-set(tgt-map) in
20.             (
21.               for all ents ∈ entities do
22.                 let body : Message-Body = ents in
23.                   def mk-Handler-Result(-,-) =
```

```
24.                     Remove-Entity([mk-Msg-Element(⟨ARGUMENT⟩,
25.                                mk-Remove-Entity-Args(body).entity-name)],ρ) in
26.                  skip;
27.           );
28.           def dummy = set-termination(Terminate-PUT) in
29.             skip;
30.           new-tgt-state.run-state := ⟨LOADED⟩;
31.           return mk-([mk-Msg-Element(⟨RESPONSE⟩,
32.                         [PROGRAM-INITIALISED])],ρ)
33.         )
34. )
```

The *Program-Under-Test* operation represents the system of interest to the human user.

```
functions
Program-Under-Test : () -o-> [token]
Program-Under-Test() ≜
(
  nil  – not relevant to this model.
)
```

The *set-registers* operation initialises the register dump of the co-routine's *saved context.* When a transfer of control takes place from the Monitor co-routine to the PUT co-routine the values of the saved context are loaded into the actual registers of the target processor. The program commences at the position indicated by *initial-pc* with the dynamic workspace position used for procedure activations and local variables indicated by *initial-sp.* The *set-registers* operation is also used to establish initial contexts of the idler, interrupt, trap and monitor co-routines in the next section.

```
operations
set-registers : Address × Address × Address -o-> ()
set-registers(reg-dump-area,initial-pc,initial-sp) ≜
(
  – target-dependent
)
```

The *set-termination* operation ensures that a successful termination of the PUT will force a return to the monitor. How this is established is both target-dependent and implementation-dependent. One possible implementation is to provide some code that is the last executed by the PUT at the bottom of the stack. *Terminate-PUT* ensures that the reason for termination is established in addition to ensuring that control is passed back to the monitor.

```
operations
set-termination : (() → [token]) -o-> B
set-termination(term-op) ≜
(
  skip; – target-dependent
  return true
)
```

*Terminate-PUT* never returns to its caller due to the fact that it transfers control back to the monitor co-routine. The first message processed by the monitor when it returns is the PROGRAM-TERMINATED message.

```
functions
Terminate-PUT : () → [token]
Terminate-PUT() ≜
1. (
2.     stop-PUT()
3. )
operations
stop-PUT : () -o-> [token]
stop-PUT() ≜
1. (
2.     let mk-Env(_,root-map,_,_) = ρ_root in
3.       let mk-Env(tgt-ws,_,_,_) = root-map(TARGET) in
4.         let mk-Workspace(_,_,old-tgt-state) = tgt-ws in
5.         (
6.           dcl new-tgt-state : Target-State := old-tgt-state;
7.           new-tgt-state.run-state := ⟨TERMINATED⟩;
8.           def term-msg = mk-Msg-Element(⟨RESPONSE⟩,
9.                                         [PROGRAM-TERMINATED]) in
10.              Push(term-msg);
11.           def dummy = transfer-to-coroutine(monitor-context.entity) in
12.              skip;
13.           return nil
14.        )
15. )
```

*Establish-Finalisation* is used to establish the finalisation code entry point that will be jumped to when the PUT terminates abnormally.

```
operations
Establish-Finalisation : Argument × Workspace -o-> Data × Handler-Status
Establish-Finalisation(args,tgt-ws) ≜
1. (
2.     let mk-Workspace(_,_,tgt-state) = tgt-ws in
3.       let mk-Establish-Finalisation-Args(final) = Message-Value(args) in
4.       (
5.         dcl new-tgt-state : Target-State := old-tgt-state;
6.         new-tgt-state.put-finalisation := final;
7.         return mk-([FINALISATION-ESTABLISHED],⟨SUCCESS⟩)
8.       )
9. )
```

*Continue-Program* only "succeeds" if the *run-state* is currently LOADED or STOPPED. In this case a transfer of control from the monitor co-routine to the PUT co-routine takes place meaning that this handler does not return until either an abnormal return is effected through the trap mechanism (this includes break events) or through the normal termination path described above. In any other *run-state*, the handler returns immediately.

```
operations
Continue-Program : Argument × Workspace -o-> Data × Handler-Status
Continue-Program(args,tgt-ws) ≜
1. (
2.     let mk-Workspace(_,_,old-tgt-state) = tgt-ws in
3.     (
4.       dcl new-tgt-state : Target-State := old-tgt-state;
5.       if new-tgt-state.run-state = ⟨NO-PUT⟩ then
6.       (
7.         return mk-([NO-PROGRAM-LOADED],⟨FAILURE⟩)
```

```
8.        )
9.        elseif new-tgt-state.run-state ∈ {⟨LOADED⟩,⟨STOPPED⟩} then
10.         (
11.           new-tgt-state.run-state := ⟨RUNNING⟩;
12.           def dummy = transfer-to-coroutine(⟨PROGRAM-UNDER-TEST⟩) in
13.             skip;
14.           return mk-([CONTINUING],⟨SUCCESS⟩)
15.         )
16.       elseif new-tgt-state.run-state ∈ {⟨RUNNING⟩,⟨TERMINATING⟩} then
17.         (
18.           return mk-([CANNOT-CONTINUE-ALREADY-RUNNING],⟨FAILURE⟩)
19.         )
20.       else
21.         (
22.           return mk-([PROGRAM-ALREADY-TERMINATED],⟨FAILURE⟩)
23.         )
24.     )
25. )
```

## Monitor Initialisation

The Idler co-routine is responsible for boot-strapping the monitor from a cold start or reset situation. The operations performed by such initialisation are divided into the following six stages:—

(1) *Idler Magic*—performs any necessary target-dependent operations to establish the Idler co-routine from a cold start.

(2) *Trap Initialisation*—performs any initialisation trap co-routine in addition to attaching to all trap producing sources.

(3) *Interrupt Initialisation*—performs any necessary attachment of the interrupt co-routine to interrupt producing sources.

(4) *Device Driver Initialisation*—establishes the default channel of communication through which the monitor receives requests made by the debug agent and other monitors.

(5) *Monitor Initialisation*—this refers to any initialisation that has to be performed by the monitor *before* a connection is established.

(6) *Enable Interrupts*—perform any target-dependent operations to enable interrupts on the target processor.

The Idler co-routine is the main program; it is effectively the monitor. The operations performed at bootstrap time to construct a valid Idler co-routine environment are both *device-dependent* and *implementation-dependent*. Anything performed at or before this stage is magic!

```
operations
bootstrap-idler : () --o--> Context
bootstrap-idler() ≜
(
  is not yet specified – target-dependent
)
```

The *initialise-bootstate* operation performs any "global" state initialisation to reset the reason code, to clear the co-routine stack, initialise the storage heap and clear the handler table of any attached co-routines.

```
operations
initialise-bootstate : () -o-> ()
initialise-bootstate() ≜
(
   current-reason := ⟨NOT-A-REASON⟩;
   current-msg := mk-Raw-Context(_,⟨IDLER⟩,nil,nil,nil);
   previous := ⟨IDLER⟩;
   cr-stack := [];
   initialise-heap();
   monitor-initialise();
   handlers := {↦};
)

initialise-heap : () -o-> ()
initialise-heap() ≜
(
   skip; – implementation-dependent
)

monitor-initialise : () -o-> ()
monitor-initialise() ≜
(
   msg-queue := [];
   user-buffer := nil;
   buffer-empty := true;
)
```

NOTE The following operations invoke monitor namespace handlers and as such use $\rho_{root}$to access the namespace hierarchy. It will be remembered that $\rho_{root}$is a component of global state and as such is usable to all monitor co-routines.

Co-routine initialisation works by having the Idler establish the co-routine with a subsequent transfer of control to the newly created co-routine so that it can perform its own initialisation. Once each co-routine has performed its initialisation, the co-routine transferred to transfers control to the Idler until each co-routine has been initialised. The Idler then waits for messages to arrive.

```
operations
initialise-coroutines : Env -o-> ()
initialise-coroutines() ≜
1.  (
2.     trap := create-coroutine(trapCR-init-pc,trapCR-init-sp,trapCR-reg-dump-base,
3.                              ⟨TRAP⟩,Trap-Coroutine);
4.     def dummy = transfer-to-coroutine(⟨TRAP⟩) in skip;
5.     interrupt := create-coroutine(intCR-init-pc,intCR-init-sp,intCR-reg-dump-base,
6.                              ⟨INTERRUPT⟩,Interrupt-Coroutine);
7.     def dummy = transfer-to-coroutine(⟨INTERRUPT⟩) in skip;
8.     initialise-driver();
9.     monitor := create-coroutine(monitorCR-init-pc,monitorCR-init-sp,
10.                             monitorCR-reg-dump-base,
11.                             ⟨MONITOR⟩,Monitor-Coroutine);
12.     def dummy = transfer-to-coroutine(⟨MONITOR⟩) in skip
13. )

create-coroutine : Address × Address × Address × Coroutine × (() -o-> ()) -o-> Context
create-coroutine(pc,sp,reg-dump,coroutine,proc) ≜
1.  (
2.     set-registers(reg-dump,pc,sp);
3.     def mk-Coroutine-env(_,_) = initialise(coroutine,proc) in
```

```
4.        return mk-Context(pc,sp,coroutine,reg-dump)
5.  )
```

The *initialise-coroutines* operation establishes the default channel of communication through *initialise-driver* so that debug agent and monitor can connect to each other at some later time. Note that failure either to initialise the device driver or to open the read and write channels respectively results in the *shutdown* of the monitor. The behaviour of the shutdown is *implementation-dependent.* The *shutdown* operation *never* returns.

```
operations
initialise-driver : Env -o-> ()
initialise-driver() ≜
1.  let mk-Env(_,root-map,_,_) = ρroot in
2.     let mk-Env(_,ext-map,_,ext-finder) = root-map(EXTERNAL) in
3.        let ρdriver = ext-map(DEFAULT-DRIVER),
4.            ρext = root-map(EXTERNAL) in
5.           let mk-Env(_,_,_,drv-finder) = ρdriver in
6.              let initialise-msg : Message = [mk-Msg-Element(⟨NAME⟩,INITIALISE)] in
7.                 def mk-Handler-Result(result,_) = drv-finder.fun(initialise-msg,ρdriver) in
8.                 (
9.                    if Message-Type(result(1)) = ⟨ERROR-DATA⟩ then
10.                   (
11.                      shutdown();
12.                   )
13.                   else
14.                   (
15.                      skip;
16.                   )
17.                   let ow-msg = [mk-Msg-Element(⟨NAME⟩,OPEN-WRITE),
18.                                 mk-Msg-Element(⟨ARGUMENT⟩,
19.                                 mk-Open-Write-Args(⟨DEFAULT-DRIVER⟩,
20.                                                    mk-token("NULL-ADDRESS")))] in
21.                      def mk-Handler-Result(ow-result,_) = ext-finder.fun(ow-msg,ρext) in
22.                      (
23.                         if Message-Type(open-write-result(1)) = ⟨ERROR-DATA⟩ then
24.                         (
25.                            shutdown();
26.                         )
27.                         else
28.                         (
29.                            skip;
30.                         )
31.                      );
32.                   let or-msg = [mk-Msg-Element(⟨NAME⟩,OPEN-READ),
33.                                 mk-Msg-Element(⟨ARGUMENT⟩,
34.                                 mk-Open-Read-Args(DEFAULT-DRIVER))] in
35.                      def mk-Handler-Result(or-result,_) = ext-finder.fun(or-msg,ρext) in
36.                      (
37.                         if Message-Type(open-read-result(1)) = ⟨ERROR-DATA⟩ then
38.                         (
39.                            shutdown();
40.                         )
41.                         else
42.                         (
43.                            skip;
44.                         )
45.                      )
46.                 )
```

*shutdown* : () $\xrightarrow{o}$ ()
*shutdown*() $\triangleq$
(
  skip – target-dependent
)

The final stage of initialisation enables interrupts for the target processor. For interrupt driven channels this means that messages can now be received from the outside world and traps/exceptions may now be caught.

operations
*enable-interrupts* : () $\xrightarrow{o}$ ()
*enable-interrupts*() $\triangleq$
(
  skip – target-dependent
)

## Summary

This chapter has developed decompositions of implicit specifications from the principle functions and operations of the model. Fuller details of other decompositions can be found in the computer files documented in the Annexes.

| Decompositions Presented in this Chapter | Decompositions Presented in the Annexes |
|---|---|
| Core Namespace Handlers (partial) | Core Namespace Handlers |
| Entity Namespace Handlers (partial) | Entity Namespace Handlers |
| Break Event Handler (partial) | Break Event Handler |
| Break Event Establishment Handlers (partial) | Break Event Establishment Handlers |
| Execution control handlers (complete) | Execution control handlers |
| Monitor Initialisation (complete) | External Namespace Handlers |
| | Communication Device Namespace Handlers |
| | Device Namespace Handlers |
| | Messaging Primitives |
| | Environment Support (Coroutines, Memory Allocation etc.) |
| | Monitor Coroutines |
| | Monitor Initialisation |
| | Message Reification (as discussed in Chapter 9) |

# Chapter 9

# Modelling Data Communication

This last component of the model is required in order to completely define interaction in terms of protocols and data formatting, between the functional core adapter (the monitor) and the debug agent.

a. Protocols must be defined for successful data transfer.

b. Bit patterns need to be assigned to meanings, i.e. encoding and decoding rules must be defined.

This chapter considers a refinement based on the ISO 8802-3 Link Layer protocol whose most common use is in the definition of a "raw" Ethernet packet.

### Issues of Heterogeneity

In order to define a concrete message format, communications media constraints must be considered; debug agent and monitor machine architecture features must also be considered. This means that issues of byte ordering, alignment and word size have to considered by any *wire protocol.*

In the case where the debug agent/monitor server and PUT exist on one machine (traditional single target, same host debuggers) there is no problem as the machine architecture is common and therefore no ambiguity should occur. Some systems that provide a cross debugging facility only allow debugging for a target of the same architecture as the host system.

The debug agent is constrained to that particular hardware platform. If the target platform changes then the debug agent must change. This method works well if the target systems are of the same kind but it is impractical for a distributed heterogeneous system where $n$ debug agents are required for each *different* target machine.

The namespace approach advocated by this thesis specifically allows *one* debug client/agent to debug a program running on a number of target machines of *differing* machine architectures *simultaneously.* The debug agent needs access to descriptions of hardware and software objects, the format of which may be used as templates to describe the namespace's physical characteristics and interface. This form of interface is desirable for heterogeneous distributed environments since target-dependencies do not have to be encapsulated within the debug agent program source text.

The wire protocol could be implemented in a number of ways. There are three alternatives to consider:—

(1) Having a common byte ordering discipline used by the debug agent. Each monitor would then be responsible for interpreting the message stream by converting to a native form.

(2) Letting the debug agent send message values in *machine native* form. This implies that any conversion to the correct native form understandable by the monitor would have to be done by the debug agent.

(3) Using a hybrid scheme in which the message tags and model handler arguments are sent in machine independent form and the reply message values are sent in native form.

The hybrid scheme has been chosen since it minimises the amount of conversion that needs to be done by the monitor. Message tags must allow individual message elements to be discriminated from each other. The message tag component of the *Msg-Element* type must therefore be in an agreed format recognised by all monitors. Because user provided *Work-Handlers* have arbitrary arguments from the point of view of the monitor, it would be computationally expensive to convert *Msg-Element* values in the monitor. All conversions which may be necessary are **required** to be performed by the *debug agent.* Arguments for model handlers, however, are described in a machine independent form.

Byte ordering is however, not the only point to be considered; machine data size must also be taken into consideration. Most of today's machines support word sizes that are a multiple of eight bits. This fits within an *octet* used by the communications hardware. There are some machines whose machine word is a non-integral number of octets, e.g. some machines have 12-bit or even 36-bit word sizes. While such machines are considerably rarer today, the protocol should be able to handle such quantities by the use of padded octets. In the case of a 12-bit word, two 8-bit octets would be required with some zero padding in the last four bits of the most significant 8-bit octet. The use of a *spare bit* tag (described later) allows for data items that are not an integral multiple of octets in size.

### Communication Media Constraints

A message, regardless of its internal structure is eventually sent across a channel as a sequence of octets. If a message value is ten bits in length then two octets will need to be used to represent the quantity in order to align the data for communication purposes. Construction and interpretation of the message value contents is the debug agent's responsibility. If the debug agent receives a reply containing a target's dump of memory then the debugger has the choice of using the raw reply data or possibly performing some further re-mapping for the purposes of "human consumption".

For media requiring a *synchronous transmission* protocol, messages are transmitted as *frames.* A frame transmission requires the beginning and end of the message to be delimited through the use of framing sequences to separate one message from another. The delimiters are known as *framing octets*, special values used to allow the communications hardware to synchronise itself to an incoming message stream. Delimiter values imply that the frame has a length. This length is typically not an arbitrary value and generally is a number of octets determined by the physical characteristics of the device. Ethernet for example has a maximum length of 1536 octets, Token Ring has a somewhat larger maximum frame size of 2048 octets while Arc-Net has a frame size of 528 octets. These maximum lengths are known as *Maximum Transmission Unit* (MTU) size. If a message is longer than the maximum frame size then a method known as message *fragmentation* allows a message to be split over a number of frames while retaining the concept that the message is logically still whole. Higher level protocols such as TCP allow message fragmentation with subsequent reconstruction at the receiver.

The wire protocol defined in this chapter permits no message fragmentation. Each message can therefore be treated as an independent *datagram.* The length of a message is limited to the MTU size of the communications medium. For example, a message sent using the Ethernet would then have a maximum length of 1536 octets.

As a message is composed of a sequence of *Msg-Elements*, the invariant on whether or not the message is too big to send though a given channel type will be determined by the sum of the

lengths of all message elements within a message being less than the MTU size for that device. The lengths of requests and replies can be determined by the *debug agent.* A reply that is too long cannot occur at the monitor end because the debug agent that generated the request is **required** by this definition to ensure that both request and eventual reply have lengths less than or equal to the MTU size for that communications medium.

The only namespace handlers where this may be a problem are those that operate on target resources such as memory or possibly even the CPU register file. If a debugger requires a memory image of 5 kilobytes to be written to the target memory namespace and the communications medium used is Ethernet then the debug agent must construct at least four separate messages to that namespace to stay within the MTU size.

The *Send/Sendto* handlers are responsible for determining whether a message to be sent is too long for the communications medium to handle. Namespace requests to target resources will typically be shorter than the replies generated from those requests. The *debug agent* must therefore ensure that such conditions do *not* occur in the first place. It must also ensure that the size of the reply that a given request can generate is no greater than the MTU. This information can be used to "tailor" requests in such a way that replies will not cause fragmentation.

## Message Structure

The wire protocol used to describe messages between debug agent and monitor has been designed to be both extensible and efficient. Efficiency is provided by minimising the amount of conversion that needs to be performed per message by the monitor. Extensibility of the protocol comes through the use of the service naming method—the namespace model. The message *denotes* the service to be invoked by explicitly naming it and allowing the finders of the namespace engine to access it. The explicit naming of a resource allows *dynamic protocol extension*—access to new services within new namespaces.

The messages sent between debug agent and monitor are characterised by regular form and structure. A *Message* is composed of a sequence of *Msg-Elements.* Each message element consists of a tag denoting the type of message element and an associated value. For a NAME tag, the value represents some token denoting the name, for an ARGUMENT tag the value represents the argument pack passed to the handler being invoked.

A request produced by the debug agent is a qualified name denoting the path to that resource and the handler to be invoked followed by the parameters to be passed to that handler.

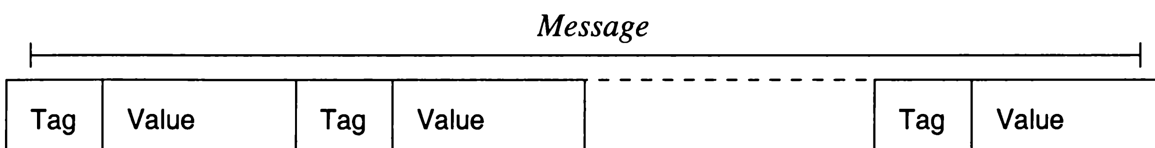



Fig. 9.1 *A namespace message is a sequence of message elements*

There are two ways by which messages originate, requests by the *debug agent* or *events* generated by either the target itself or other target monitors. Where the message comes from is irrelevant. The form each message takes is the same.

The following BNF message grammar describes the format of *requests* expected to be sent to the monitor. Non-terminals are italicised, terminal symbols (individual *Msg-Elements*) are in a sans serif font:—

*Request* → *External-Name-Msg* | *Name-Msg* | *External-Break-Msg* | *User-Data-Msg*
*External-Name-Msg* → *External-Name-Path* *Name-Path* Argument
*External-Name-Path* → External-Name
*Name-Msg* → *Name-Path* Argument
*Name-Path* → Name$^+$
*External-Break-Msg* → External-Break
*User-Data-Msg* → User-Data

A message to a namespace is a name path followed (optionally) by any arguments to the handler being invoked. There may only be *one* argument pack associated with any handler invocation. The external break and user data messages are processed by the interrupt co-routine. User data is filtered so that the PUT can use it. The external break is processed by the interrupt co-routine so that the external break occurrence is treated as a trap message which halts the PUT as detailed in Chapter 6.

There are two types of reply, successful and unsuccessful. The *ERROR* and *RESPONSE* message kinds are the result of a handler failing or the invocation of a handler failing. The *ERROR-DATA* and *RESPONSE-DATA* message kinds are produced by *result bearing* handlers. *Work-Handlers* or *Env-Handlers* produce data results through these message kinds. The BNF grammar describing the expected message reply formats is:—

*Reply* → *Response-Msg* | *Data-Response-Msg* | *Error-Msg* |
*Data-Error-Msg* | *User-Data* | *External-Break*

*Response-Msg* → Response
*Data-Response-Msg* → Response-Data
*Error-Msg* → Error
*Data-Error-Msg* → Error-Data
*User-Data* → User-Data
*External-Break* → External-Break

Unlike the *External-Name-Msg* and *Name-Msg*, the reply contains only a *single* message element. The value of the message element is in the native form of the responder. It is up to the debug agent and debug client as to whether the response data is left raw or converted in some way.

## Transmission Packets

Each *conc-Msg-Element* is self identifying through a tag type. The length of the element value in octets is determined by a field in the tag prefix. The length in octets delimits message elements from each other. The total size of a message element is the size of the tag plus the number of octets (determined by the tag) used by that value.

The size of the tag and its representation is based upon the practical requirements of the message and the medium it travels along. A tag of eight bits would mean that the number of octets in size that a value could be is the difference in the number of bits required to represent the message type (four bits) plus the spare bit used when the target architecture's word size is not based on an integral number of octets. This leaves three bits to be used as the octet count. A message value of size eight octets is insufficient for representing values of any significance. However, a tag size of sixteen bits gives eleven bits of range for the octet count. This means that message values could be up to 2047 octets in length. This exceeds the maximum size of an Ethernet frame. As

a request typically consists of a number of elements it is normally the case that the MTU size of the communications medium is exceeded before the octet count of a message element is exceeded in the situation where Ethernet is used as the communications media.

If the spare bit is set for a tag, the argument pack for a particular namespace handler requires an argument bitstream. When the spare bit is detected, the *third* octet in the message element (independent of the octet count) is used to determine the number of bits spare when the value is packed to an exact number of octets. For example, if the argument size were ten bits, a single value for that target machine would have the spare bit in the tag set, have the octet count set to *two* and have the bits spare octet set to six bits. It is up to each implementation of the monitor software to extract these values from the octet stream.

The following types define the message bitstream at the bit and octet level. *conc-Messages* and *conc-Msg-Elements* are composed of octet streams.

$$
\begin{aligned}
&\mathit{Binary\text{-}Digit} = \\
&\qquad \mathbf{N} \\
&\qquad\quad \mathsf{inv}\ num \ \triangleq (num = 0) \lor (num = 1);
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Bit\text{-}Sequence} = \\
&\qquad \mathit{Binary\text{-}Digit}^{+};
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Octet} = \\
&\quad \mathit{Binary\text{-}Digit}^{+} \\
&\qquad \mathsf{inv}\ seq \ \triangleq \\
&\qquad\quad \mathsf{len}\ seq = 8;
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Tag} = \\
&\quad \mathit{Octet}^{+} \\
&\qquad \mathsf{inv}\ seq \ \triangleq \\
&\qquad\quad \mathsf{len}\ seq = 2;
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Octet\text{-}Count} = \\
&\qquad \mathbf{N} \\
&\qquad\quad \mathsf{inv}\ cnt \ \triangleq \\
&\qquad\qquad (cnt > 0) \land (cnt \leq 2047);
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Spare\text{-}Bits} = \\
&\qquad \mathbf{N} \\
&\qquad\quad \mathsf{inv}\ cnt \ \triangleq \\
&\qquad\qquad (cnt \geq 1) \land (cnt \leq 7);
\end{aligned}
$$

The *tag* component of a *conc-Msg-Element* is two octets in length. The width of the *Type-Field* is four bits, the *spare* bit is one bit and the *Size-Field* is eleven bits. An eleven bit value for the octet count gives a range of up to 2047 octets (this exceeds the Ethernet MTU), excluding the spare bits octet that follow if the spare bit is set in the tag. This gives considerable freedom to describe arguments or names depending on how these values are refined. The *Spare-Bits* type denotes the number of bits used to pad to an exact number of octets if quantities are not an integral multiple of octets in size.

$$
\begin{aligned}
&\mathit{Type\text{-}Field} = \\
&\qquad \mathit{Bit\text{-}Sequence} \\
&\qquad\quad \mathsf{inv}\ seq \ \triangleq \\
&\qquad\qquad \mathsf{len}\ seq = 4;
\end{aligned}
$$

*Size-Field* =
  *Bit-Sequence*
    inv *seq* $\triangleq$
      len *seq* = *11;*

The *Tag-Data* type represents the tag components from a more abstract viewpoint that reflects the model specified. This is the level at which any further refinement would be done to extract or construct a given *conc-Msg-Element.*

*Tag-Data* ::
  *kind* : *Message-Kind*
  *sz* : *Octet-Count*
  *sp* : **B**
    inv *data* $\triangleq$
      let *overlay* = *mk-Tag-Bits(t-fld,s-fld,sp-fld)* be st
      *((t-fld = kind)* $\wedge$ *(s-fld = sz)* $\wedge$ *(sp-fld = sp))* in
        *data* = *overlay;*

The *Tag-Bits* type is a bit-wise description of the *Tag-Data* type given above. One could imagine the *Tag-Bits* as being *superimposed* over the *Tag* type, something that is implied by the invariant for the *Tag-Data* type.

*Tag-Bits* ::
  *type* : *Type-Field*
  *size* : *Size-Field*
  *spare* : *Binary-Digit;*

The *Spare* and *Spare-Data* types are defined in a similar way to the *Type-Field* and *conc-Message-Kind* types. The *Spare* octet holds the number of bits short of a full octet at the end of a *conc-Message-Value.* The appearance of this octet in the message stream immediately after the tag is contingent upon whether the spare bit field has been set. If set, the *Spare* octet appears in the message stream, otherwise message data appears immediately after the tag.

*Spare* =
  *Octet;*

*Spare-Data* =
  *Spare-Bits;*

A *conc-Message-Body* in its most fundamental form is a sequence of octets whose length and form is described by the *Tag* type and *Spare* type. The "value" contained within the *conc-Message-Body* is based upon the message *kind* described in the tag. For example, in a received message element this could be a NAME, EXTERNAL-NAME, ARGUMENT, EXTERNAL-BREAK or USER-DATA. Where the message element is being sent this could be an EXTERNAL-BREAK, USER-DATA, RESPONSE, ERROR, DATA-RESPONSE or DATA-ERROR.

*conc-Message-Body* =
  $Octet^{+}$ |
  $\vdots$ ; – also for formatted types such as argument packs.

A *conc-Msg-Element* consists of the tag, the spare octet (if necessary) and the message value associated with a tag of that type. The maximum length of a *conc-Msg-Element* is the sum of the octet count, length of the tag and *spare bits* octet—a total of 2050 octets. It will be remembered that this total is in excess of the Ethernet MTU. Message parsing is straightforward due to the tag delimiting the data it describes.

*conc-Msg-Element* ::
*tag* : *Tag-Bits*
*spare* : *[Spare]*
*value* : *conc-Message-Body*
inv *element* $\triangleq$
let *mk-conc-Msg-Element(tg,_,val)* = *element* in
let *mk-Tag-Bits(_,size,_)* = *tg* in
len *val* = *retr-Nat(size)*;

Message tag byte ordering has to be consistent across implementations of monitors for different targets. Message tag byte ordering is necessary because the tag is sixteen bits—how this value is *addressed* by a given machine can affect its interpretation of where the type, octet count and spare bit fields lie. This can result in a type being incorrectly determined if the byte ordering is *big endian* or *little endian.* Since the type information is of primary importance to the monitor the storage of type information is most efficient in the four least significant bits of the sixteen bit tag. Masking operations can then be used to determine the message type. The spare bit indicator is the most significant bit of the sixteen bit tag—a bit test can be used to establish its (rare) existence. The octet count is extracted by right shifting operations. These operations are most easily accomplished if the tag is transmitted in *little endian* form, ie., low octet-high octet order.

The first of the following two diagrams displays the tag left to right from most significant bit to least significant bit to show the structure. The second diagram shows the structure of the tag as it would be *transmitted* across a communications medium.

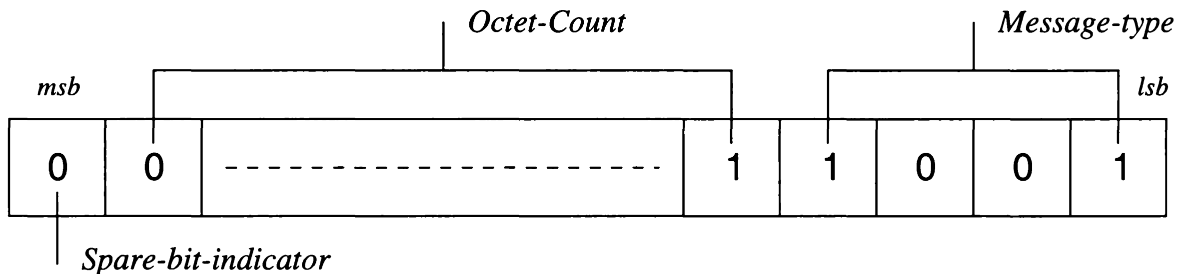



Fig. 9.2 *Raw tag format*

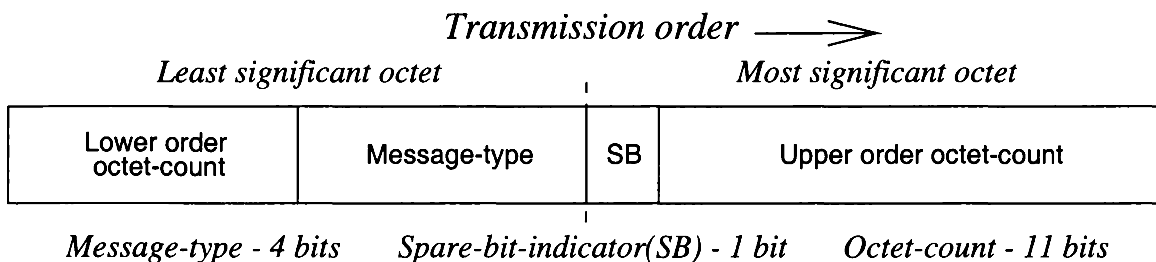



Fig. 9.3 *The wire representation of the message element tag*

The message value component of a *conc-Msg-Element* follows a message tag if *spare bit* is not set, otherwise it follows the *spare* octet used to determine how many bits of the last octet represent padding bits that are *not* part of the message value. The length of the message value in octets is determined by the *octet-count* value part of the tag.

Name and external message elements must have an associated value (the octet count is necessarily non-zero). For example, a namespace must have a name! However, it is possible to have

handlers that do not require arguments, examples include the *Open-Session* and *Close-Session* handlers of External namespace.

The following diagrams consider two situations, the first is the transmission of values when the target architecture has a word size that is based upon a multiple of eight bits. Such a situation will normally mean that the *spare-bit-indicator* will be cleared and the message value will immediately follow the tag. The second situation considers a target architecture whose word size is not a multiple of eight bits, in this case the quantities are twelve bit words. The first case has an octet count of three, the second has an octet count of five.

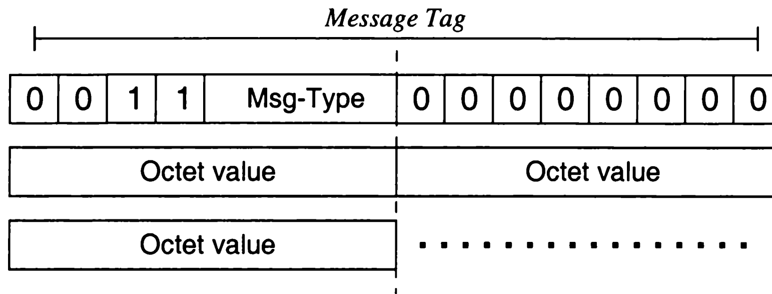



Fig. 9.4 *A three byte message value encoded as part of a message element value*

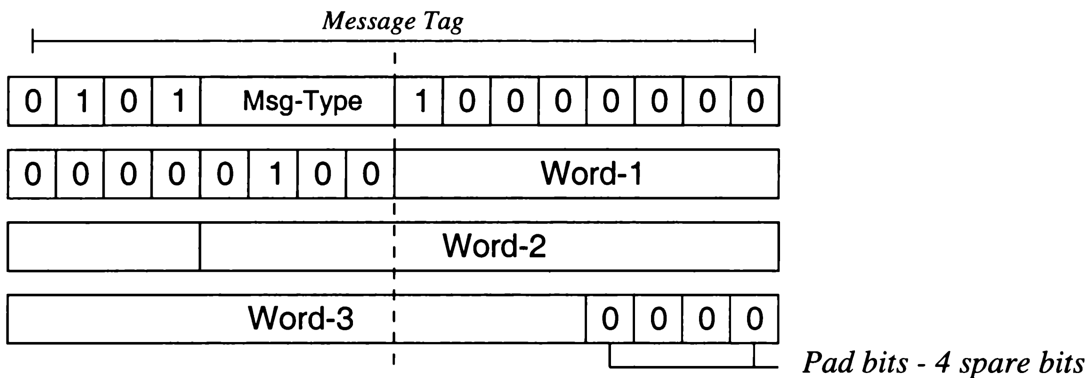



Fig. 9.5 *Three twelve bit word values encoded as part of a message element value*

Note that in the above diagram there are 4 "spare" bits used to pad to the next multiple of octets. The most efficient means to package thirty-six bits in octet form is to pack the values into forty bits (five octets).

## Machine Independent Tag Values

The discussion so far has implied that message values are transmitted in native form. This is only true for those *Work-Handlers* that are *not* part of the formal model described in this thesis. Any types associated with the formal model *require* an agreed upon concretion. The reason for this is that if the debug agent is target-independent then it must communicate uniformly irrespective of the target to which it is connected. The converse of this would be to require that there be multiple debug agents which is undesirable.

There are two kinds of concretion required for model types:—

(1) The types whose only defined operation is that of equality testing need only have an agreed concretion based on the number of octets required to express that *bit pattern*.

(2) Types whose operations not only include equality testing but possibly other arithmetic operations. These types not only require an agreed concretion based upon the number of octets required to represent the *value* but also a target independent representation that specifies the ordering and significance of the quantity. Note that all such values are *unsigned.*

The values associated with the message tag kinds EXTERNAL-NAME, EXTERNAL-BREAK and NAME are of the former variety. The values associated with the tag type ARGUMENT required by handlers within the model (eg. setting or modifying a break) are an example of the latter (although some arguments types may also be of the former kind).

Values associated with the tag types NAME, EXTERNAL-NAME and EXTERNAL-BREAK have to be transmitted in target-independent form to the extent that the size in octets needs to be defined. Arguments used by the *Env-Handlers* and *Work-Handlers* in the model also need to be given some target-independent concretion. For example, the *Count* type used to set breaks must not only have its size defined but also its representation well defined. For similar reasons the workspace size required by *Add-Name* requires an agreed representation.

The general rule is:—

> Those types associated with the model need a defined concrete representation so that debug agents can communicate with arbitrary targets. Those handlers that are not part of the model have *argument* values passed to them in *native form.*

This means that concretions of message values to a form meaningful to the monitor **may** need to be performed before some of the values of these model types can be used by the monitor. It is the concretions of these *model types* that will be defined in this section.

Such concretions give an upper bound in terms of the overhead required to convert these types to native form. Like the tag definitions in the previous section, model types whose use requires a target-independent concretion eg. numeric and address types are transmitted in *little endian* byte order—least significant octet to most significant octet.

This implies that NAME, EXTERNAL-NAME and EXTERNAL-BREAK tags **never** have the *spare* bit set, this will also be the case for ARGUMENT tags to *Work-Handlers* defined in the model. At worst the overhead is the *possible* swapping of a few octets, this is of course target-dependent. If the machine is *little endian,* while the values of these model types might need *truncation,* no values need to be swapped.

The model types that require some standard concretion are:—

a. *conc-Address*—this is the target pointer type used to denote a location in the address space of the target. All addresses map to physical locations in the address map. A value of this type is typically passed within an ARGUMENT message element. The size of an object of this type is eight octets.

b. *conc-Workspace-Size*—used by *Add-Name* to provide a workspace for the new environment created. The workspace is used for maintaining the state, handler code and map of that namespace. A value of this type is passed within an ARGUMENT message element, the size of an object of this type is eight octets.

c. *conc-Name*—denotes the name of a given namespace. Such values occur in both NAME tags and as ARGUMENT values. A value of this type is passed within a NAME message element and an ARGUMENT value. The size of an object of this type is six octets.

d. *conc-External-Name*—the same as immediately above. The size of an object of this type is therefore also specified to be six octets.

e. *Breakpoint types*—these are used by the model to set, clear and handle break events.

(1) *conc-Break-Address*—this type represents an address at which a breakpoint is set. It is synonymous with the *conc-Address* type. A value of this type is supplied as part of an ARGUMENT message element value. The size of an object of this type is eight octets.

(2) *conc-Machine-Address*—this type represents a machine network address used to specify where a message is to be sent. A value of this type is supplied as part of an ARGUMENT message element value. The size of an object of this type is six octets.

(3) *conc-Count*—this type represents a value used by the break handler to provide counting breakpoints. A value of this type is supplied as part of an ARGUMENT message element value. The size of an object of this type is four octets. This value for concretion purposes is in *little endian* form.

(4) *conc-Predicate-Data*—this type is an address used by the entity predicate to make assertions upon. A value of this type is supplied as part of an ARGUMENT message element value. The size of an object of this type is eight octets.

(5) *conc-Break-Id*—the size of an object of this type is five octets and is used to identify a breakpoint. A value of this type is supplied as part of an ARGUMENT or EXTERNAL-BREAK message element value.

(6) *conc-Break-Kind*—the size of an object of this type is one octet and is used to denote the type of breakpoint to be set, a local, group or global break. A value of this type is supplied as part of an ARGUMENT or EXTERNAL-BREAK message element value.

While arguments to handlers which are a part of the model require appropriate concretions, model errors and responses also require appropriate concretions. These concretions are not to be confused with RESPONSE-DATA and ERROR-DATA whose values are transmitted in native form. The values associated with ERROR and RESPONSE have a standard concretion like the above types that is known to both monitor and debug agent.

(1) *conc-Error*—the size of an object of this type is 1 octet. A value of this type is associated with an ERROR *conc-Msg-Element.*

(2) *conc-Response*—the size of an object of this type is 1 octet. A value of this type is associated with a RESPONSE *conc-Msg-Element.*

The following type definitions describe the concretions used to represent the model types. The remaining definitions are described in the Annex F.

values

$Name\text{-}Type\text{-}Size : \mathbf{N} =$
$6;$

types

$conc\text{-}Name =$
$Octet^{+}$
$\mathsf{inv}\ seq \triangleq$
$\mathsf{len}\ seq = Name\text{-}Type\text{-}Size;$

The break identity type concretion must also be agreed upon by all monitors; the message value format is fixed. The break identity as described in Chapter 4 consists of a name and break kind. The break identity is 5 octets in size, giving a large range of possible break identities. The reasons behind the size of the break identity being so large are to do with the fact that entity predicates are *named* in terms of their associated break entry. This means breaks are named in the *same* way as

namespaces, this consistency is of considerable importance when considering an implementation. If name values are the same size, an implementation that deals with these names will not have to cope with any special cases. It will remembered that the abstract form of *Break-Identity* described previously was:—

*Break-Identity* ::
  *kind* : *Breakpoint-Kind*
  *id* : token

A concretion of this type in terms of octets sent would be:—

types
  *conc-Break-Identity* =
    $Octet^+$
      inv *b-id* ≙
        len *b-id* = *Break-Identity-Type-Size;*
  *conc-Break-Kind* =
    $Octet^+$
      inv *b-kind* ≙
        len *b-kind* = *Break-Kind-Type-Size;*
  *conc-Break-Message* ::
    *b-kind* : *conc-Break-Kind*
    *b-id* : *conc-Break-Identity;*

The following diagram describes the format of an external breakpoint message. The first octet of the message value describe the breakpoint kind followed by the last 5 octets describing the breakpoint identity, which may either be group or global.

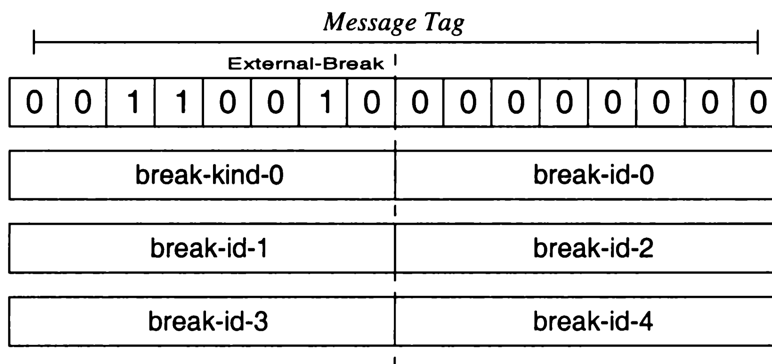



Fig. 9.6 *The contents of an external breakpoint message*

Message retrieval operations are responsible for converting a stream of octets into a meaningful *conc-Message* consisting of a sequence of *conc-Msg-Elements.* These operations are described in the following sections.

## Concrete Message Format

When a message is sent to or received from a channel it is just a sequence of octets. This section formalises the concrete message format used by the protocol being defined between monitor and debug agent.

The concrete message format is based upon the ISO 8802-3 Ethernet packet. This does not mean that communicating devices are restricted to Ethernet hardware for communication between debug agent and monitor server to take place. This frame format need not be used just for Ethernet,

serial line devices can also make use of the packet format defined. Only *one* message packet format is defined here. This simplifies the design of drivers which have the task of retrieving and setting values that are *device-dependent.*

This packet structure definition specifies the data component of a message packet format which is assumed to begin (and possibly end) with appropriate framing for transmission purposes.

```
types
    conc-Machine-Address =
                        Octet+
                           inv m-addr ≜ len m-addr = Machine-Address-Type-Size;
    conc-Raw-Msg-Length =
                        Octet+
                           inv pk-len ≜ len pk-len = 2;
    Raw-Packet ::
            destination : conc-Machine-Address
            source : conc-Machine-Address
            length : conc-Raw-Msg-Length
            message : conc-Raw-Packet
               inv mk-Raw-Packet(destination,source,length,raw-message) ≜
                   let mk-conc-Raw-Packet(_,_,_,data) = raw-message in
                      let bseq : Bit-Sequence = octet-to-bits(length) in
                         (retr-Nat(bseq) = Raw-Packet-Header-Size + len data) ∧
                         (destination ≠ source);
    conc-Raw-Packet ::
               dest-port : Octet
               src-port : Octet
               control : Octet
               data : conc-Raw-Message;
    conc-Raw-Message =
                   Octet+;
```

The above *Logical-Packet* structure is suggested if network connections are shared with other communications protocols. The *dest-port* and *source-port* are used to indicate protocol port identities. This can be used to enforce a protocol type scheme so that the debug protocol can be discriminated from other protocols which may be using the communications medium. In the case of an isolated target development system, there might only be one protocol being used in the network, the debug protocol. If the development environment being used is within a network that includes both target systems and host systems then the target systems on which the monitor servers reside will only accept packets of the right type from the debug agent and other monitors. The same applies to host machines on the network.

As a message arrives from the communication device used by the monitor it is in *Raw-Message* form. The namespace model only uses a *Message* type for encapsulating the tagged *Msg-Element* descriptors. Operations used to *Send* and *Receive* messages must convert from a Raw-Message to a Message when receiving or a Message to a Raw-Message when sending. There are two design approaches to consider:—

(1) Retrieval/concretion at the driver level.

(2) Retrieval/concretion at the input-output (Send/Sendto/Receive) *Work-Handler* level.

There are reasonable arguments in support of either alternative. In the former case it could be argued that drivers perform device-dependent operations, one of these operations should be

the retrieval or concretion of messages. The approach taken here, however, is the latter view. This has the benefit of localising retrieval and concretion functions to the *abstract* device driver interface of External namespace. If retrieval and concretion takes place as part of the input-output *Work-Handlers* then drivers need only perform device-dependent operations to send and receive messages. The advantage of the second approach is that it simplifies the implementation of device driver operations.

### Message Construction and Extraction—Retrieval Functions

Retrieval functions are used to map between their "realised" form in an implementation and the *abstract model types.* Such realised forms may vary considerably in structure from the abstract model definition although it is important that the type invariants associated with the abstraction are upheld by the concretion. This is demonstration of the *adequacy* of the concretion to reflect the abstraction. Inverse mappings also exist and as a result there are also reverse retrieval functions that take the abstract form and produce the concretion.

The abstract model sees a message as a stream of tokens, an actual implementation takes this representation and produces a concretion—an octet stream to be transmitted by the communications hardware. In the other case, the implementation receives an octet stream with a retrieval function producing a *conc-Message* constructed of individual *conc-Msg-Elements* suitable for consumption by the monitor.

### Message Retrievals

The process of raw message retrieval takes a concretion and maps this to the corresponding model abstraction. In the case of the raw message an octet stream is converted to the model messages as described in Chapters 4 and 5.

Message retrieval has to consider the mappings required by the concretions of types to their model counterparts. Unsigned numeric quantities are "converted" (if necessary) to the form required by the model for a given implementation. It will be remembered that numeric types are little endian. For target implementations where the target's byte ordering is big endian, conversion needs to be performed by the target.

The *retrieve-Message* operation takes a stream of octets and constructs a model message from it.

```
functions
retrieve-Message : conc-Raw-Message → Message
retrieve-Message(raw-msg) ≜
1. (
2.    let mk-(msg,msg-ok) = retr-Message(raw-msg,mk-Msg-Element(⟨NAME⟩,⟨ROOT⟩)) in
3.      if msg-ok then
4.        msg
5.      else
6.        mk-Message(mk-Msg-Element(⟨ERROR⟩,⟨BAD-MESSAGE-FORMAT⟩))
7. )
```

The retrieval of a message is the retrieval of its constituent elements. The concretions of each element are self describing allowing each message element to be easily retrieved.

```
functions
retr-Message : conc-Raw-Message × Msg-Element → [Message] × B
retr-Message(raw-msg,previous) ≜
1. (
```

```
2.     if len raw-msg = 0 then
3.        mk-(nil,true)
4.     else
5.        let mk-(length,element) = retr-Msg-Element(raw-msg,previous) in
6.           if element = nil then
7.              mk-(nil,false)
8.           else
9.              let used ⌒ rest = raw-msg in
10.                let used ⌒ rest : conc-Raw-Message be st len used = length in
11.                   let mk-(msg,result) = retr-Message(rest,element) in
12.                      if result then
13.                         mk-(msg ⌒ [element],result)
14.                      else
15.                         mk-(nil,false)
16. )
```

The use of *prev*, the previously retrieved message element assists in maintaining message well-formedness as described earlier by the message grammar. The rules are:—

a. Only *sensible* messages can be retrieved. This means that EXTERNAL-NAME, NAME, EXTERNAL-BREAK and USER-DATA messages are legal while RESPONSE, RESPONSE-DATA, ERROR, ERROR-DATA and ARGUMENT packs by themselves are not.

b. The message has to be parsed according to the following rules:—

(1) An EXTERNAL-NAME *can* be followed by a NAME or ARGUMENT *Msg-Element*.

(2) An EXTERNAL-BREAK *cannot* have any other *Msg-Element* following it.

(3) A NAME *can* be followed by a NAME, an ARGUMENT *Msg-Element* or nothing.

(4) USER-DATA *cannot* have any other *Msg-Element* following it.

(5) An ARGUMENT *cannot* have any other *Msg-Element* following it

c. Depending on the model handler, argument packs *may* have to be converted to native form for consumption by the model. Addresses and numeric types such as counts may have to undergo this conversion.

```
functions
retr-Msg-Element : conc-Raw-Message × Msg-Element → N × [Msg-Element]
retr-Msg-Element(raw-msg,previous) ≜
1. (
2.     if len raw-msg < 3 then
3.        mk-(len raw-msg,nil)
4.     else
5.        let rest ⌒ tag = raw-msg in
6.           let rest ⌒ tag : conc-Raw-Message be st len tag = 2 in
7.              let mk-(length,kind,pad,err) = unhatch-tag(tag) in
8.                 if err then
9.                    mk-(length,nil)
10.                else
11.                   if length > len rest then
12.                      mk-(length,nil)
13.                   else
14.                      if pad then
15.                         let padbits ⌒ value ⌒ tail = rest in
16.                            let padbits ⌒ value ⌒ tail : conc-Raw-Message be st
17.                               (len padbits = 1) ∧ (len value = length) in
18.                               let padcnt : N = retr-Nat(octet-to-bits(padbits)) in
19.                                  let val ⌒ padding = value in
```

```
20.                        let val ⌒ padding : conc-Raw-Message be st len padding =
padcnt in
21.                            mk-(length + 3,rx-filter(prev,length,kind,value))
22.         else
23.           let value ⌒ tail = rest in
24.             let value ⌒ tail : conc-Raw-Message be st len value = length in
25.               mk-(length + 2,rx-filter(prev,length,kind,value))
)
```

The *unhatch-tag* function takes a tag and extracts any tag information required by the retrieval, the *Message-Kind*, value length and padding indicator.

The filtering function *rx-filter* performs well-formedness checking and possible type conversions for model handlers. The well-formedness checking performed amounts to consideration of whether the previous element in the message constitutes a valid prefix as outlined above. Checking argument packs is more involved but essentially relies on the well-formedness of the pack as constructed by the debug agent or monitor. Some argument packs can have their length checked because their argument packs are of a fixed, expected size. Others such as *Add-Handler* are more difficult due to the variable size of the pack.

The result of *rx-filter* is a *conc-Message.* This is a message that consists of a single *conc-Msg-Element.* A message is then just a concatenation of singular messages which are *conc-Msg-Elements* as per the original definition.

The checking of arguments is done for *all* model handlers, user defined/target-dependent handlers such as those that manipulate target hardware resources *do not* have their arguments checked by the retrieval function. It is up to the debug agent to produce an argument pack that can be used by the particular *Work-Handler* concerned.

Argument checking may seem to indicate great computational cost. In practice this cost is minimal. Only certain types are actually converted by re-ordering words, the numeric and address types are the only instances. Depending on the target architecture, such conversions may not need to be done although it is possible that address truncation may need to be performed for some implementations.

Argument checking at retrieval time makes the most sense and makes the rest of the monitor considerably less complicated as a result. The two other options are impractical and severely limit the ability of the model:—

a. *Have the finder of each environment examine the argument pack*—This complicates the implementation of the *Find-Name* handler, which has to become environment *specific.* What is worse with such a method is that dynamic addition of handlers becomes impossible, the finder must change to reflect the newly added handler(s).

b. *Have each handler check its arguments*—This is even less practical because the handler has to be written in a particular way. Desirably, *Work-Handlers* should be written and appear to behave as local procedure calls.

Argument checking during retrieval allows early detection of messages that should not invoke a corresponding handler. These include messages from the debug agent that invoke handlers that should only be invoked due to events occurring. An example of this is the break handler—the debug agent should not be able to invoke it! If the debug agent produces such messages they should be discarded by the monitor. The following handlers *cannot* be invoked by the agent and will be discarded by Check-Arguments as such:—

(1) ADD-ENTITY, REMOVE-ENTITY and FIND-ENTITY are only used by the PUT and monitor respectively. It is useless to have the debug agent invoke these.

(2) The input-output handlers should not be able to be called by the agent, especially RECEIVE but also SEND and SENDTO. If their invocation by the debug agent were allowed, strange behaviour could occur, especially with RECEIVE—deadlock could occur!

```
functions
Check-Arguments : N × Name × conc-Message-Body → [Msg-Element]
Check-Arguments(pack-length,prev-name,pack) ≜
1.  (
2.     cases prev-name:
3.     ⋮ – other model handler arguments
4.     ⟨INIT-PROG⟩ →
5.        if pack-length = Address-Type-Size + Address-Type-Size +
6.                         Address-Type-Size + Workspace-Type-Size then
7.           mk-Msg-Element(⟨ARGUMENT⟩,retr-Initialise-Program-Args(pack))
8.        else
9.           nil,
10.    ⋮ – other model handler arguments
11.    others → – pass non model handler arguments through untouched
12.       mk-Msg-Element(⟨ARGUMENT⟩,retr-tokens(pack))
13.    end
14. )
```

User defined or target specific handlers have their arguments passed in native form—no conversion is done. The same is true for responses generated by such handlers. With the exception of handlers that cannot be invoked by the debug agent, the model handlers described have their arguments converted to the target native form by the *rx-filter* operation. Given the little endian bias taken by the concretion of this protocol, little endian machines need not do any further conversion other than possible truncation eg., 8 octet addresses being converted to 4 octet (32 bit) addresses.

The following operation is an example of an argument conversion for the *INIT-PROG* model handler:—

```
functions
retr-Initialise-Program-Args : conc-Message-Body → Message-Body
retr-Initialise-Program-Args(raw-pack) ≜
1.  (
2.     let mk-conc-Initialise-Program-Args(c-initpc,c-initsp,c-wsbase,c-wssize) = raw-pack in
3.        let initpc : Address = {retr-Addr(octet-to-bits(c-initpc)) ↦ undefined},
4.            initsp : Address = {retr-Addr(octet-to-bits(c-initsp)) ↦ undefined},
5.            wsbase : Address = {retr-Addr(octet-to-bits(c-wsbase)) ↦ undefined},
6.            wssize : N1 = retr-Nat(octet-to-bits(c-wssize)) in
7.        mk-Initialise-Program-Args(initpc,initsp,wsbase,wssize);
8.  )
```

## Argument Type Conversions

Argument type conversions are performed for each model handler that can be invoked by the debug agent. It is important to note that handlers locally invoked by event handlers such as the interrupt and trap co-routines have their argument packs constructed in native form and *do not* undergo the conversion process that is required by messages received from the debug agent.

Various *target independent* types such as names and network addresses are *bit patterns* that require no conversion. Only the address and numeric types used by a given target are converted

at retrieval time. The operations presented are target-dependent. If no conversion is required (the machine is little endian) then an implementation of these operations *may* not need to perform anything.

functions
*retr-Addr : Bit-Sequence* → **N**

*retr-Count : Bit-Sequence* → **N**

*retr-Nat : Bit-Sequence* → **N**

## Message Concretion

Concretion functions map the abstract model types to a "real" representation—in this case the form the values take over a communications channel.

The process of message concretion is somewhat simpler than that of retrieval. There are two reasons for this:—

(1) The number of message classes is smaller. A concretion takes results of the model to be sent to a debug agent or another monitor. These are restricted to external break, response, error and user-data messages. These "replies" are collectively realised as a single *conc-Msg-Element* which results in none of the parsing that was required as part of retrieval.

(2) The model produces correct messages for the appropriate situation. A concretion can reasonably assume that the data used to construct a message is correct. Conversely, retrieval functions have to be able to handle malformed messages.

Despite these two differences, concretion can still be considered to be the *inverse* of retrieval. In this situation, the concretion results in an octet stream that can be used for monitor–monitor and monitor–debug-agent communication.

Message concretion (the *conc-Message* operation) takes a *Message* (which is really just **one** *Msg-Element* because it is a reply) and concretes this into an octet stream.

functions
*conc-Message : Message* → *[Octet⁺]*
*Message-conc(model-msg)* ≜
*1.* (
*2.*     if len *model-msg* = *1* then
*3.*         *conc-MsgElement(*hd *model-msg)*
*4.*     else
*5.*         nil
*6.* )

Message element concretion (the *conc-Msg-Element* operation) takes a *Msg-Element* and constructs a *Tag-Bits* prefix and octet sequence representing the message value. In the case of response data and user-data the values are returned to the client in native form, no conversion is done to the octet stream. This means less overheads are required by the monitor to return data to the debug agent. The monitor only has to consider model messages such as errors and responses in terms of a particular concretion.

In a mechanism that is similar to message retrieval, the message concretion operation has a *tx-filter* operation. It is considerably simpler due to there being no need to analyse handler arguments as *rx-filter* did. With the exception of model messages arising from errors or responses, no conversion has to be done by the filter.

### Errors Concretions

Errors are treated in a similar manner to names. Each model error has an associated concretion with errors from user defined handlers being disjoint from those of the model. The error concretion is a one octet value, all model errors being mapped to their respective values at concretion time.

*Errors* =
⟨*ARGUMENT-EXPECTED*⟩ |
⟨*ATTEMPTED-TO-CLOSE-UNOPENED-CHANNEL*⟩ |
⟨*BAD-MESSAGE-FORMAT*⟩ |
⟨*CANNOT-DISPATCH-MSG-TYPE*⟩ |
⟨*CANNOT-FREE-HANDLER*⟩ |
⟨*CANNOT-FREE-OLD-HANDLER*⟩ |
⟨*CANNOT-FREE-WORKSPACE*⟩ |
⟨*CANNOT-ROLL-BACK*⟩ |
⟨*CLOSE-READ-FAILED*⟩ |
⟨*CLOSE-WRITE-FAILED*⟩ |
⟨*DID-NOT-REPLACE*⟩ |
⟨*DRIVER-NOT-FOUND*⟩ |
⟨*EXPECTED-NAME*⟩ |
⟨*INCOMPLETE-PATH*⟩ |
⟨*INSUFFICIENT-ARGS*⟩ |
⟨*INVALID-ENTITY*⟩ |
⟨*NAME-ALREADY-EXISTS*⟩ |
⟨*NAME-NOT-FOUND*⟩ |
⟨*NO-BREAK-AT-ADDRESS*⟩ |
⟨*NO-ENV-HANDLER-ARGS*⟩ |
⟨*NO-MORE-WORKSPACE*⟩ |
⟨*NOT-A-WORK-HANDLER*⟩ |
⟨*NOT-AN-ENV*⟩ |
⟨*OPEN-READ-FAILED*⟩ |
⟨*OPEN-WRITE-FAILED*⟩ |
⟨*UNHANDLED-INTERRUPT*⟩;

a. ARGUMENT-EXPECTED is returned if the *Msg-Element* was expected but not encountered for this handler invoked.

b. ATTEMPTED-TO-CLOSE-UNOPENED-CHANNEL is returned if an attempt was made to close a channel that was not currently open.

c. BAD-MESSAGE-FORMAT is returned by a handler in which the expected message format is different from that expected by that handler.

d. CANNOT-DISPATCH-MESSAGE-TYPE is returned by the monitor if the dispatcher cannot dispatch a tag.

e. CANNOT-FREE-HANDLER is returned by *Remove-Handler* and *Replace-handler* if the *Workspace* containing the handler within a given namespace cannot be freed for future use.

f. CANNOT-FREE-OLD-HANDLER is returned by *Replace-Handler* when the handler that existed previously cannot be removed.

g. CANNOT-FREE-WORKSPACE is returned by *Remove-Namespace* when the workspace that contains an environment is unable to be freed for future use. The environment still exists if this is the case.

h. CANNOT-ROLL-BACK is returned by *Close-Read* and *Close-Write* when an attempt to restore read and write handles to their previous values fails because the previous values are

invalid. This can only actually occur the first time the monitor cold-starts so there can be *no* previous read and write handles.

i. CLOSE-READ-FAILED is returned when an attempt to close the previous input channel fails.

j. CLOSE-WRITE-FAILED is returned when an attempt to close the previous output channel fails.

k. DID-NOT-REPLACE is returned by *Replace-Handler* if it was unable to replace the current Work-Handler by a newly supplied Work-Handler.

l. EXPECTED-NAME is similar to ARGUMENT-EXPECTED and is returned when a name *Msg-Element* is expected within the current message context but not present.

m. INCOMPLETE-PATH is returned when the path parsed by a *Find-Name* is insufficient to invoke either an Env-Handler or Work-Handler.

n. INSUFFICIENT-ARGUMENTS is returned when the number of arguments required by a particular Work-Handler is not satisfied.

o. INVALID-ENTITY is returned when an attempt to add an entity determines that the context pointer lies outside that entity's workspace.

p. NAME-ALREADY-EXISTS is returned by either *Add-Name* or *Add-Handler* if the name or handler to be added already exists in that namespace environment.

q. NAME-NOT-FOUND is returned by either *Remove-Name* or *Remove-Handler* if the name or handler to be removed does not exist in that namespace environment.

r. NO-ENV-HANDLER-ARGS is returned when arguments required by an Env-Handler are not present. An Env-Handler's first component of its argument pack is a name type representing the environment.

s. NO-MORE-WORKSPACE is returned by *Add-Name* when the environment to be added cannot be added.

t. NOT-A-WORK-HANDLER is returned *Remove-Handler* when the name being removed does not refer to a Work-Handler but an environment. *Remove-Handler* cannot remove environments.

u. NOT-AN-ENV is returned by *Remove-Name* if the name associated with an environment is actually not an environment.

v. INVALID-MESSAGE is returned in the case where the message tag does not match with any of the defined *Message-Kinds.*

w. UNHANDLED-INTERRUPT is returned by the interrupt co-routine in the situation where an interrupt is not one of the three specified event sources.

The following error concretion function signature takes an error kind and produces an octet concretion if the reply *Msg-Element* is of kind ERROR. For full details, refer to Annex F.

$$conc\text{-}Error : Errors \rightarrow Octet$$

### Response Concretions

There are two kinds of response, those that return status or data. *Work-Handlers* return RESPONSE-DATA, *Env-Handlers* may return either in addition to any errors. Status responses provide an indication of the success of a particular action in a manner similar to errors described in the previous section. RESPONSE-DATA messages return any information and status for the invocation of that *Work-Handler* in native form. A RESPONSE message describes the results of the model handlers.

Much like the concretion used to describe errors, responses also require a one octet concretion for the code.

*Responses* =
*⟨CLOSE-READ-SUCCESSFUL⟩* |
*⟨CLOSE-WRITE-SUCCESSFUL⟩* |
*⟨HANDLER-ADDED-OKAY⟩* |
*⟨HANDLER-REMOVED-OKAY⟩* |
*⟨HANDLER-REPLACED-OKAY⟩* |
*⟨NAME-ADDED-OKAY⟩* |
*⟨NAME-REMOVED-OKAY⟩* |
*⟨OPEN-READ-SUCCESSFUL⟩* |
*⟨OPEN-WRITE-SUCCESSFUL⟩;*

a. CLOSE-READ-SUCCESSFUL indicates that the channel previously used for receiving messages was deactivated successfully.

b. CLOSE-WRITE-SUCCESSFUL indicates that the channel previously used for sending messages was deactivated successfully.

c. HANDLER-ADDED-OKAY indicates that a handler was successfully added to an environment. Henceforth, that handler is available for use within that environment.

d. HANDLER-REMOVED-OKAY indicates that a handler was successfully removed from an environment. Henceforth, that handler is unavailable for use within that environment.

e. HANDLER-REPLACED-OKAY indicates that a handler of a given name was replaced by a another handler. While the name used to denote the handler remains the same, the handler semantics themselves may change.

f. NAME-ADDED-OKAY indicates that an environment subordinate to that in which the name was added has been created.

g. NAME-REMOVED-OKAY indicates that an environment subordinate to that of the parent has been removed along with any other environments attached to that child environment.

h. OPEN-READ-SUCCESSFUL indicates that the channel that was requested for read use has been opened and communication will take place through that channel from then on.

i. OPEN-WRITE-SUCCESSFUL indicates that the channel requested for write use has been opened and communication will take place through that channel from then on.

The following response concretion function signature takes a response kind and produces an octet concretion if the reply *Msg-Element* is of kind RESPONSE. For full details, refer to Annex F.

*conc-Response* : *Responses* $\rightarrow$ *Octet*

## Summary

This chapter has described the packet format for a datagram based interaction between debug agent and monitor. The packet format is based upon an ISO 8802-3 Ethernet packet although it may be applied to *any* communications medium. How this packet is transmitted is dependent upon the communication medium.

A model of the device independent communication drivers has produced *retrieval* operations, *concretion* operations and value specifications. Retrieval operations map a concrete representation of data to an abstract form used by the model. Conversely, a concretion operation takes model values and produces a concrete representation of the data.

Model handlers may require type concretions to be performed by the monitor. Errors and replies from model handlers result in particular concretions of model values being sent back to the debug agent. In the case of handlers that are not a part of the model, arguments are passed to the handler in native form as are replies back to the debug agent.

Minimum *essential* requirements for a debug agent implementation have been derived as a result of the packet format and the decisions made earlier in the design of the architecture:—

a. The debug agent will require descriptor information for the target monitor and hardware to which it is connected. The descriptor information content allows symbolic access to namespace resources. The set of descriptors grows or shrinks based upon the addition or removal of namespaces.

b. The debug agent is responsible for the construction of requests whose length does not exceed the MTU. In doing this, the protocol is simplified for the monitor and debug agent.

An exhaustive list of requirements is given in Annex G.

# Chapter 10

# Conclusions

A formal model suitable for debugging distributed embedded systems has been described. This chapter evaluates the contributions made by this work within the field of distributed debugger technology, evaluates any deficiencies and discusses future work.

The original motivations for this research arose from my perception that the tools used for development of embedded systems were inadequate to cope with the demands of today's distributed systems.

Current tools are deficient because only parts of the problem are addressed. In many current tools a separate debug agent has to be associated with each target. Many debug agents can neither control the execution of a distributed system nor have the ability to capture global state essential for distributed debugging. Other systems get the distributed debugging capabilities correct but at the cost of flexibility.

None of debugger architectures described in the literature can dynamically adapt to *change*. Debuggers suitable for the debugging of desktop systems where hardware (and software) configuration is static are not nearly as applicable to the dynamic environment of an embedded system, especially a distributed one.

Embedded systems are by their nature dynamic, reactive systems whose software and hardware changes over time. The ability to dynamically alter parts of the monitor is essential for systems whose existence relies upon continuous operation; this is particularly the case where dynamic addition or removal of hardware is concerned.

### Contributions

I believe that this thesis has made the following contributions to the fields of debugging tools:—

(1) A *formal* model of a debug server for distributed systems has been defined.

(2) The use of the *namespace* to model distributed embedded systems hardware and software.

There is a pervasive culture suggesting that while the process of debugging is an "art", the construction of essential development tools to assist in the debugging process such as debuggers and linkers is a still blacker art. This thesis has demonstrated through formal modelling that debug servers share many design characteristics similar to typical event driven programs—event queues, event dispatch and processing. The emphasis on low-level code and multiple threads of control are concessions that need to be made by an implementation for a given target. Above this low-level target-dependent code, the essential event-loop structure of the monitor and constituent co-routines is shared by all implementations.

After data structure refinement, the specification given becomes a rubber stamp for all real implementations. A deliberate choice has been made in this model definition *not* to further refine any structures such as sets, maps and composite data objects given in the chapters describing the

model. The appropriate data structures are realisable in many ways, the structures chosen in a refinement to represent those of the model will be decided based on typical time-space trade-off criteria. The specification provides a useful behavioural view in which the matter of "how" becomes irrelevant.

Development of a few low-level target-dependent values, types and operations has been deliberately avoided. While a small amount of extra work is needed to completely specify any particular implementation it is considered that to do so here would detract from the usefulness of this work as an abstraction.

This thesis provides a model useful for the implementation of *extensible* monitors and debug agents. The *namespace* model incorporated provides a basis upon which all monitor services are implemented.

Viewing configuration through the namespace model offers several advantages that are a product of the ability to extend the monitor at run-time.

a. The monitor always has an optimal *working set* of handlers, the monitor only requires an interface to those namespace objects present. In this manner, the monitor is always at its optimum size. Handlers for a namespace only need to satisfy the the capabilities of that namespace.

b. The namespace model is *secure.* The debug agent is constrained to the interface specified by the namespace model.

c. There are no restrictions placed on the PUT. The PUT has full control of the target system (to the extent where even the monitor's workspace may be corrupted).

d. The ability to *modify* monitor behaviour—except for altering core namespace functionality—*Env-Handlers* cannot be modified.

e. Because namespaces and their respective handlers are *dynamically linked* there is no need to power-down the system to add or remove monitor functionality.

The dynamic configuration allowed at run-time by the namespace model *requires* debug agent support. The design of most debug agents is currently limited to the target machines for which they will be used. This requires the modification of debug agent source code each time a debugger is constructed—so called *static retargetting.* The namespace model takes a configuration oriented view of the target. When namespaces are added to or removed from the monitor it is imperative that the debug agent be notified of these changes and update itself accordingly. Current debug agents are designed for static configurations of the target system. The implication here being that current debug agent design is *unsuitable* for the demands of dynamically configuring distributed (embedded) systems.

An implementation of a monitor based upon the model presented in this thesis requires a different approach to debug agent design. With careful design taking into account the requirements described in Annex G, the debug agent can also become *dynamically retargettable.* The requirements of capturing consistent global state has resulted in a particular debugging architecture—the single debug agent, multiple monitor approach.

Restriction of current debug agent design has been imposed by the "hard-wiring" of the debug agent to a particular target architecture. The namespace model specified here, however, can be used to avoid this hard-wiring. Namespaces make the construction of heterogeneous distributed debug agents possible by allowing the separation of access methods (handlers) from their possible symbolic representation on the debug agent.

If a debug agent implements multiple sessions then each session could provide a target-dependent view of each connection between debug agent and monitors. Each namespace on the target would have a corresponding description at the debug agent. This description can be used to *describe* the namespace and the handler interface for the physical instantiation of that namespace on the target system. The namespace descriptions can be used by the debug agent to check outgoing requests to the monitor. The ability to check request contents means that little or possibly no checking of arguments needs to be done by the monitor. Debugger design would then be a matter of designing an appropriate user interface suitable to co-ordinate the information from each session into information to be used by the human user to assist in distributed debugging.

The namespace model is used *only* by the monitor/debug-agent. It is not a general purpose facility for use by the PUT. The PUT can use target resources in any way appropriate to the requirements of the application. Four entry points can be used by the PUT, two are used for entity registration and deregistration, the remaining two for primitive input-output between debug agent and monitor. These entry points indirectly use the the namespace model to perform their operations.

## Limitations

Being able to extend the monitor is useful but there are restrictions on what can sensibly be replaced or removed. As a general rule, care must be taken when replacing or removing handlers that may be invoked by an event occurring locally on the target upon which the monitor resides. Event handlers for traps and interrupts cannot be replaced, they are not part of the namespace model but they do *produce* messages that are *consumed* by the model. In practice this will mean that handlers such as the *Break-Handler* can be replaced but the replaced handler *must* have the same calling interface as before. This will also be the case for *all* handlers which can be invoked by local event occurrences.

The datagram approach to protocol design minimises the state that needs to be maintained; each message is independent of any other. The problem with such an approach is that it limits the size of the message to the MTU of the communications medium. The concretions defined in Chapter 9 place a maximum limit on the size in octets of the message content to be 2047 octets. Clearly, depending upon the medium used, the name path-argument form of a namespace request restricts the size of handlers that can be added to a given environment.

It is important to point out that the namespace model does not limit the operations that can be performed by an environment, it only provides a useful basis for abstraction and extension of behaviour. There is nothing preventing a handler from behaving erroneously by failing to terminate or performing completely "useless" operations—just as there is nothing to stop the implementer of a linked list class from adding a method that displays "Hello world!". Any *Work-Handler* can be added; whether that handler is useful is entirely another matter.

## Future Work

Future work based upon this thesis can be divided into two areas:—

(1) Further derivation and development of the model presented in this thesis with an aim toward an implementation.

(2) The design of the debug agent and debug client.

It has been found in the course of the work described in this thesis that VDM-SL does not seem to provide all of the semantics which the dynamism of the model requires. Since this thesis was

first drafted, an object-oriented extension to VDM-SL, called VDM++ [IF++99], has emerged. While at first sight this could have been a better formalism to use, in my opinion it would still be difficult to reconcile the co-routine and namespace entities in a unified VDM-style formal model description. Having said this, the use of the VDM-SL notation to clarify thinking has been most useful; it has provided a basis for logical reasoning about the namespace model, something which would not have been possible without the use of a formal modelling notation. The availability of the IFAD VDM-SL Toolbox [IFAD99] has enabled me to syntax check my notation and, to a large extent, type-check the model. It seems as if further mathematical modelling research may be needed prior to being able to complete a fully formalised denotational definition of the name-space model.

The VDM-SL Toolbox also provides facilities to produce C++ code from explicit operations and functions developed using the tool. Because the toolbox also contains both model checking and testing facilities to "execute" an explicit operation or function, the code generation phase should be considered as a final step with only target-dependencies being required to be implemented. Using this mechanism it could be possible to construct "virtual" debug test harnesses to test the debug server against a primitive debug agent that produced messages. This ignores the fact that debug agent and debug client issues (described below) need to be addressed.

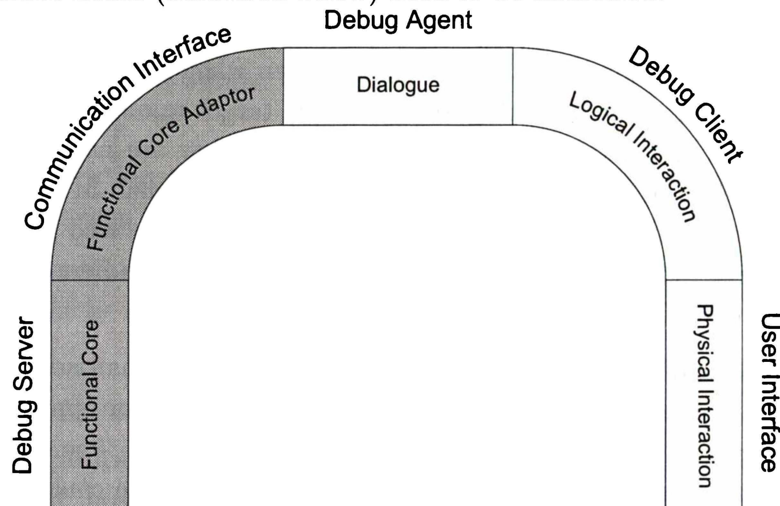



Fig. 10.1 *Arch-Slinky revisited—Future Directions*

The emphasis of this thesis has been on the design of the monitor, namespace model and protocol used to provide debugging functionality. There has been some mention of the debug agent but the design issues regarding the debug agent were not a part of this thesis.

As can be seen by the above diagram this thesis has considered only the shaded area, the debug server side. The debug agent is effectively a mirror image of the monitor to which it is connected. The debug agent is an external producer of messages, the protocol is a request-reply interaction. The debug agent will provide symbolic translation capabilities that map debug client requests to a form acceptable to the namespace handlers.

To offer a solution to the challenge of what a debug client user interface should provide the best means of looking at the problem is to ask the question—what is required to debug a distributed system? One of the problems of distributed debugging is the large state space produced by the execution of such systems. Given the support for task relative breakpoints, group/global breaks and the "traditional" local break mechanisms, there needs to be a way of displaying the associations between tasks that comprise the PUT. These tasks may be part of another session the debug agent has with another target involved as part of the PUT. Graphical techniques to manage such

complexity are not out of the question but such a solution is only one of many ways in which a debug client could be designed.

### Summary

The research aims posited at the start of this thesis have been achieved. The original question considered a debugging architecture suitable for heterogeneous distributed embedded systems. The namespace model derived provides support for this as well as the ability to extend behaviour dynamically at run-time.

# Annex A

# Namespace Model Definition

The *VDM-SL* contents of Annexes A through E are provided on a CD-ROM attached to the inside back cover of this thesis. The CD-ROM is provided in the standard ISO-9660 format and as such can be mounted and read by any operating system capable of mounting such a file system. Most operating systems that support a CD-ROM also support the ISO-9660 file system format, Windows NT/2000/9x, Linux and other Unix dialects should work without problems.

### CD-ROM Contents

The CD-ROM contains four directories:—

(1) packages—contains GhostScript and GhostView packages for installation on Windows NT/2000/9x so that the thesis PostScript files can be viewed and printed.

(2) thesis—contains PostScript document source that may be printed to a PostScript printer or through some other non-PostScript printer via GhostView or GhostScript.

(3) vdm—contains the files arranged for checking by the *IFAD VDM-SL Toolbox.* Three IFAD VDM-SL Toolbox project (.prj) files are provided; thesis-explicit.prj, thesis-implicit.prj and thesis-wireprotocol.prj. Provided this directory is copied directly to the location where the projects were created then these project files may be used as is by opening the project file from within the IFAD VDM-SL Toolbox executable. This directory should be copied to "C:" if the provided project files are to be used without the need for modification.

(4) vdm-grouped—contains the same files as in the vdm directory above but in a form that reflects the logical groupings of operations and namespace handlers. A logical grouping consists of:—

a. The class or behaviour being modelled. This also includes global state definitions, types and values.

b. Whether the definition is given implicitly or explicitly.

c. Whether the action is a *function* or an *operation* that affects state.

It is the contents of this directory that will be used as the basis for describing the file contents of Annexes A through E.

All VDM-SL contained in the files on this CD-ROM adheres to the *VDM-SL* ASCII syntax so that it may be interpreted or checked by tools such as the *IFAD VDM-SL Toolbox.* This is in contrast to the definitions given in the chapters which use the *VDM-SL* mathematical syntax, a form that is more often used for presentation purposes. Despite the visual differences, both forms are syntactically equivalent.

The files are provided as is and have been left uncompressed. All VDM specification files have a (.vdm) extension and may be viewed with any text editor.

## The IFAD VDM-SL Toolbox

The *IFAD VDM-SL Toolbox* software used for developing and checking the models presented in this thesis may be downloaded from *www.ifad.dk*. The software is licensed. An evaluation license of the software may be obtained by e-mailing *info@ifad.dk* and requesting a license file that is limited (at the time of this writing) to one month's evaluation of the software. Versions of the software are available for Solaris, HP-UX 10.0, Linux and Windows NT/2000/9x. However, it should be noted that an evaluation license was only available for the Windows NT/2000/9x version at the time of this writing.

It should be noted that there are some minor semantic differences between the *VDM-SL Toolbox* and the ISO VDM standard. These are noted in the IFAD VDM-SL Language [IFAD99] document which is also obtainable online from *www.ifad.dk*. A description of the level of conformance to the standard attained by the VDM-SL Toolbox software is given in section 2 of IFAD VDM-SL Language [IFAD99] document. Semantic differences are given in section 16 of the same document.

## A Description of Namespace Model Core

The contents of this annex describe the core of the namespace model:—

a. The core namespace model values and types.

b. The implicitly specified core namespace model handlers, for example, *Find-Name* are presented.

| Pathname | Filename |
|---:|---:|
| vdm-grouped | types.vdm |
| vdm-grouped | values.vdm |
| vdm-grouped/base/explicit/functions | Message_Type.vdm |
| vdm-grouped/base/explicit/functions | Message_Value.vdm |
| vdm-grouped/base/implicit/operations | Find_Name.vdm |
| vdm-grouped/base/implicit/operations | Add_Name.vdm |
| vdm-grouped/base/implicit/operations | Remove_Name.vdm |
| vdm-grouped/base/implicit/operations | Add_Handler.vdm |
| vdm-grouped/base/implicit/operations | Remove_Handler.vdm |
| vdm-grouped/base/implicit/operations | Replace_Handler.vdm |

# Annex B

# Namespace Services

### A Description of Contents

The contents of this annex specify services built on top of the namespace framework presented in Annex A.

The following service namespaces are *implicitly* specified in this annex:—

a. Monitor global state definition.

b. Device namespace which includes handlers for namespaces such as register and memory namespace.

c. Target namespace which contains handlers for the controlling of PUT execution, the manipulation of break events and control of PUT entities.

| Pathname | Filename |
|---:|---:|
| vdm-grouped | states.vdm |
| vdm-grouped/cpu/implicit/operations | Get_Register.vdm |
| vdm-grouped/cpu/implicit/operations | Put_Register.vdm |
| vdm-grouped/device/implicit/operations | Add_Hardware.vdm |
| vdm-grouped/device/implicit/operations | Get.vdm |
| vdm-grouped/device/implicit/operations | Put.vdm |
| vdm-grouped/memory/implicit/operations | Get_Store.vdm |
| vdm-grouped/memory/implicit/operations | Put_Store.vdm |
| vdm-grouped/target/implicit/functions | test_modify_invariant.vdm |
| vdm-grouped/target/implicit/functions | test_invariant.vdm |
| vdm-grouped/target/implicit/functions | map_to_sequence.vdm |
| vdm-grouped/target/implicit/functions | get_entity_break_entry.vdm |
| vdm-grouped/target/implicit/operations | Add_Entity.vdm |
| vdm-grouped/target/implicit/operations | add_entry.vdm |
| vdm-grouped/target/implicit/operations | add_true_predicate.vdm |
| vdm-grouped/target/implicit/operations | Break_Handler.vdm |
| vdm-grouped/target/implicit/operations | Clear_Break.vdm |
| vdm-grouped/target/implicit/operations | clear.vdm |
| vdm-grouped/target/implicit/operations | clear_entry.vdm |
| vdm-grouped/target/implicit/operations | Continue_Program.vdm |
| vdm-grouped/target/implicit/operations | determine_break.vdm |
| vdm-grouped/target/implicit/operations | do_find_entity.vdm |
| vdm-grouped/target/implicit/operations | Establish_Finalisation.vdm |
| vdm-grouped/target/implicit/operations | Find_Entity.vdm |
| vdm-grouped/target/implicit/operations | get_entity.vdm |
| vdm-grouped/target/implicit/operations | get_entry_with_addr.vdm |
| vdm-grouped/target/implicit/operations | get_old_entry.vdm |
| vdm-grouped/target/implicit/operations | global_break_handler.vdm |
| vdm-grouped/target/implicit/operations | group_break_handler.vdm |
| vdm-grouped/target/implicit/operations | Initialise_Program.vdm |
| vdm-grouped/target/implicit/operations | local_break_handler.vdm |
| vdm-grouped/target/implicit/operations | local_break_response.vdm |

| Pathname | Filename |
|---|---|
| vdm-grouped/target/implicit/operations | Modify_Break.vdm |
| vdm-grouped/target/implicit/operations | modify_entry.vdm |
| vdm-grouped/target/implicit/operations | modify_entry_set.vdm |
| vdm-grouped/target/implicit/operations | push_add_predicate.vdm |
| vdm-grouped/target/implicit/operations | push_break_handler_msg.vdm |
| vdm-grouped/target/implicit/operations | push_continue.vdm |
| vdm-grouped/target/implicit/operations | push_error_no_break.vdm |
| vdm-grouped/target/implicit/operations | push_group_break_sendto.vdm |
| vdm-grouped/target/implicit/operations | push_remove_predicate.vdm |
| vdm-grouped/target/implicit/operations | push_replace_predicate.vdm |
| vdm-grouped/target/implicit/operations | push_saved_id.vdm |
| vdm-grouped/target/implicit/operations | Remove_Entity.vdm |
| vdm-grouped/target/implicit/operations | remove_delayed.vdm |
| vdm-grouped/target/implicit/operations | Set_Break.vdm |
| vdm-grouped/target/implicit/operations | set.vdm |
| vdm-grouped/target/implicit/operations | set_brk.vdm |
| vdm-grouped/target/implicit/operations | set_delayed_break.vdm |
| vdm-grouped/target/implicit/operations | set_entry.vdm |
| vdm-grouped/target/implicit/operations | set_finalisation.vdm |
| vdm-grouped/target/implicit/operations | set_termination.vdm |
| vdm-grouped/target/implicit/operations | Terminate_PUT.vdm |

# Annex C

# Model Drivers

### A Description of Contents

The contents of this annex presents material that was discussed in Chapter 6 and 7:—

a. Monitor co-routines, event handlers and dispatching.

b. Core primitive operations for the management of storage, co-routines and queues.

c. Session management using External namespace.

d. Driver initialisation and usage in combination with External namespace.

| Pathname | Filename |
|---:|---:|
| vdm-grouped/drivers/explicit/operations | close_read.vdm |
| vdm-grouped/drivers/explicit/operations | close_write.vdm |
| vdm-grouped/drivers/implicit/operations | Driver_Initialise.vdm |
| vdm-grouped/drivers/explicit/operations | driver_init.vdm |
| vdm-grouped/drivers/explicit/operations | get_message.vdm |
| vdm-grouped/drivers/explicit/operations | open_read.vdm |
| vdm-grouped/drivers/explicit/operations | open_write.vdm |
| vdm-grouped/drivers/explicit/operations | poll.vdm |
| vdm-grouped/drivers/explicit/operations | put_message.vdm |
| vdm-grouped/external/implicit/operations | do_close_read.vdm |
| vdm-grouped/external/implicit/operations | do_close_write.vdm |
| vdm-grouped/external/implicit/operations | do_cr.vdm |
| vdm-grouped/external/implicit/operations | do_cw.vdm |
| vdm-grouped/external/implicit/operations | do_open_read.vdm |
| vdm-grouped/external/implicit/operations | do_open_write.vdm |
| vdm-grouped/external/implicit/operations | drv_check_driver.vdm |
| vdm-grouped/external/implicit/operations | Check_Driver.vdm |
| vdm-grouped/external/implicit/operations | Check_Driver_Handler.vdm |
| vdm-grouped/external/implicit/operations | Close_Read.vdm |
| vdm-grouped/external/implicit/operations | Close_Session.vdm |
| vdm-grouped/external/implicit/operations | Close_Write.vdm |
| vdm-grouped/external/implicit/operations | Open_Read.vdm |
| vdm-grouped/external/implicit/operations | Open_Session.vdm |
| vdm-grouped/external/implicit/operations | Open_Write.vdm |
| vdm-grouped/external/implicit/operations | Receive.vdm |
| vdm-grouped/external/implicit/operations | Send.vdm |
| vdm-grouped/external/implicit/operations | Sendto.vdm |

| Pathname | Filename |
|---|---|
| vdm-grouped/global-ops/implicit/operations | Destroy-Entity.vdm |
| vdm-grouped/global-ops/implicit/operations | Register_Entity.vdm |
| vdm-grouped/global-ops/implicit/operations | UserData.vdm |
| vdm-grouped/global-ops/explicit/operations | Receive_Message.vdm |
| vdm-grouped/global-ops/explicit/operations | Send_Message.vdm |
| vdm-grouped/global-ops/explicit/operations | Sendto_Message.vdm |
| vdm-grouped/global-ops/explicit/operations | stop_PUT.vdm |
| vdm-grouped/global-ops/explicit/operations | Terminate_PUT.vdm |
| vdm-grouped/global-ops/explicit/operations | UserData.vdm |
| vdm-grouped/global-ops/explicit/operations | User-Get.vdm |
| vdm-grouped/global-ops/explicit/operations | User-Put.vdm |
| vdm-grouped/monitor/explicit/functions | map_external_break.vdm |
| vdm-grouped/monitor/explicit/operations | Deregister_Hardware.vdm |
| vdm-grouped/monitor/explicit/operations | Communication_Handler.vdm |
| vdm-grouped/monitor/explicit/operations | External_Break_Dispatch.vdm |
| vdm-grouped/monitor/explicit/operations | External_Name_Dispatch.vdm |
| vdm-grouped/monitor/explicit/operations | get_interrupt_reason.vdm |
| vdm-grouped/monitor/explicit/operations | get_trap_reason.vdm |
| vdm-grouped/monitor/explicit/operations | Idler_Coroutine.vdm |
| vdm-grouped/monitor/explicit/operations | Interrupt_Coroutine.vdm |
| vdm-grouped/monitor/explicit/operations | Monitor_Coroutine.vdm |
| vdm-grouped/monitor/explicit/operations | monitor_body.vdm |
| vdm-grouped/monitor/explicit/operations | Name_Dispatch.vdm |
| vdm-grouped/monitor/explicit/operations | process_message.vdm |
| vdm-grouped/monitor/explicit/operations | Register_Hardware.vdm |
| vdm-grouped/monitor/explicit/operations | Trap_Coroutine.vdm |
| vdm-grouped/monitor/explicit/operations | trap_dispatch.vdm |
| vdm-grouped/monitor/implicit/operations | Deregister_Hardware.vdm |
| vdm-grouped/monitor/implicit/operations | External_Break_Dispatch.vdm |
| vdm-grouped/monitor/implicit/operations | get_interrupt_reason.vdm |
| vdm-grouped/monitor/implicit/operations | get_trap_reason.vdm |
| vdm-grouped/monitor/implicit/operations | Name_Dispatch.vdm |
| vdm-grouped/monitor/implicit/operations | Register_Hardware.vdm |
| vdm-grouped/primitives/implicit/operations | Acquire_Space.vdm |
| vdm-grouped/primitives/implicit/operations | Fill.vdm |
| vdm-grouped/primitives/implicit/operations | generate_loc.vdm |
| vdm-grouped/primitives/implicit/operations | get_dump.vdm |
| vdm-grouped/primitives/implicit/operations | get_hw_info.vdm |
| vdm-grouped/primitives/implicit/operations | get_locality.vdm |
| vdm-grouped/primitives/implicit/operations | get_new_added_hw_id.vdm |
| vdm-grouped/primitives/implicit/operations | get_new_removed_hw_id.vdm |
| vdm-grouped/primitives/implicit/operations | get_position.vdm |
| vdm-grouped/primitives/implicit/operations | get_ws_chunk.vdm |
| vdm-grouped/primitives/implicit/operations | Insert.vdm |
| vdm-grouped/primitives/implicit/operations | Pop_CRStack.vdm |
| vdm-grouped/primitives/implicit/operations | Push.vdm |
| vdm-grouped/primitives/implicit/operations | Push_CRStack.vdm |
| vdm-grouped/primitives/implicit/operations | Release_Space.vdm |
| vdm-grouped/primitives/implicit/functions | map_closure.vdm |
| vdm-grouped/primitives/implicit/functions | map_code.vdm |
| vdm-grouped/primitives/implicit/functions | map_data.vdm |
| vdm-grouped/primitives/implicit/functions | target_dependent_reason.vdm |
| vdm-grouped/primitives/implicit/functions | target_dependent_interrupt_handler.vdm |
| vdm-grouped/primitives/implicit/functions | address_in_range.vdm |

| Pathname | Filename |
|---|---|
| vdm-grouped/primitives/explicit/operations | Acquire_Space.vdm |
| vdm-grouped/primitives/explicit/operations | alloc_ws.vdm |
| vdm-grouped/primitives/explicit/operations | attach.vdm |
| vdm-grouped/primitives/explicit/operations | create_coroutine.vdm |
| vdm-grouped/primitives/explicit/operations | dealloc_ws.vdm |
| vdm-grouped/primitives/explicit/operations | detach.vdm |
| vdm-grouped/primitives/explicit/operations | full_stop.vdm |
| vdm-grouped/primitives/explicit/operations | get_locs.vdm |
| vdm-grouped/primitives/explicit/operations | get_ws_chunk.vdm |
| vdm-grouped/primitives/explicit/operations | halt.vdm |
| vdm-grouped/primitives/explicit/operations | handle_interrupt.vdm |
| vdm-grouped/primitives/explicit/operations | interrupt_occurs.vdm |
| vdm-grouped/primitives/explicit/operations | Pop.vdm |
| vdm-grouped/primitives/explicit/operations | pool_locs.vdm |
| vdm-grouped/primitives/explicit/operations | Push.vdm |
| vdm-grouped/primitives/explicit/operations | Push_CRStack.vdm |
| vdm-grouped/primitives/explicit/operations | remove.vdm |
| vdm-grouped/primitives/explicit/operations | set_registers.vdm |
| vdm-grouped/primitives/explicit/operations | stop.vdm |
| vdm-grouped/primitives/explicit/operations | transfer_to_handler.vdm |
| vdm-grouped/primitives/explicit/operations | transfer_to_coroutine.vdm |
| vdm-grouped/primitives/explicit/operations | unget_ws_chunk.vdm |
| vdm-grouped/primitives/explicit/functions | address_in_range.vdm |
| vdm-grouped/primitives/explicit/functions | current_coroutine.vdm |
| vdm-grouped/primitives/explicit/functions | deallocate_locs.vdm |
| vdm-grouped/primitives/explicit/functions | initialise.vdm |
| vdm-grouped/primitives/explicit/functions | map_data.vdm |
| vdm-grouped/primitives/explicit/functions | map_code.vdm |
| vdm-grouped/primitives/explicit/functions | map_closure.vdm |
| vdm-grouped/primitives/explicit/functions | resume.vdm |
| vdm-grouped/primitives/explicit/functions | target_dependent_reason.vdm |

# Annex D

# Operational Decomposition of Core Handlers

### A Description of Contents

The contents of this annex represent the operational decompositions of core handlers presented in the earlier annexes. These include:—

a. The core namespace model handlers.

b. The target namespace specialised entity namespace handlers; *Find-Entity*, *Add-Entity* and *Remove-Entity*.

c. The target namespace break handler and break event set, modify and clear operations.

d. The target namespace execution control handlers.

e. Operations for the establishment of monitor initial state and shutdown.

| Pathname | Filename |
|---|---|
| vdm-grouped/base/explicit/operations | Add_Handler.vdm |
| vdm-grouped/base/explicit/operations | Add_Name.vdm |
| vdm-grouped/base/explicit/operations | dispatch_name.vdm |
| vdm-grouped/base/explicit/operations | do_child_remove.vdm |
| vdm-grouped/base/explicit/operations | Find_Name.vdm |
| vdm-grouped/base/explicit/operations | Remove_Handler.vdm |
| vdm-grouped/base/explicit/operations | Remove_Name.vdm |
| vdm-grouped/base/explicit/operations | Replace_Handler.vdm |
| vdm-grouped/monitor/explicit/functions | map_external_break.vdm |
| vdm-grouped/monitor/explicit/operations | Communication_Handler.vdm |
| vdm-grouped/monitor/explicit/operations | Deregister_Hardware.vdm |
| vdm-grouped/monitor/explicit/operations | External_Break_Dispatch.vdm |
| vdm-grouped/monitor/explicit/operations | External_Name_Dispatch.vdm |
| vdm-grouped/monitor/explicit/operations | get_interrupt_reason.vdm |
| vdm-grouped/monitor/explicit/operations | get_trap_reason.vdm |
| vdm-grouped/monitor/explicit/operations | Idler_Coroutine.vdm |
| vdm-grouped/monitor/explicit/operations | Interrupt_Coroutine.vdm |
| vdm-grouped/monitor/explicit/operations | Monitor_Coroutine.vdm |
| vdm-grouped/monitor/explicit/operations | monitor_body.vdm |
| vdm-grouped/monitor/explicit/operations | Name_Dispatch.vdm |
| vdm-grouped/monitor/explicit/operations | process_message.vdm |
| vdm-grouped/monitor/explicit/operations | Register_Hardware.vdm |
| vdm-grouped/monitor/explicit/operations | Trap_Coroutine.vdm |
| vdm-grouped/monitor/explicit/operations | trap_dispatch.vdm |
| vdm-grouped/monitor/implicit/operations | Deregister_Hardware.vdm |
| vdm-grouped/monitor/implicit/operations | External_Break_Dispatch.vdm |
| vdm-grouped/monitor/implicit/operations | get_interrupt_reason.vdm |
| vdm-grouped/monitor/implicit/operations | get_trap_reason.vdm |
| vdm-grouped/monitor/implicit/operations | Name_Dispatch.vdm |
| vdm-grouped/monitor/implicit/operations | Register_Hardware.vdm |

| Pathname | Filename |
|---|---|
| vdm-grouped/target/explicit/functions | entity_active.vdm |
| vdm-grouped/target/explicit/functions | entry_exists.vdm |
| vdm-grouped/target/explicit/functions | entry_with_addr.vdm |
| vdm-grouped/target/explicit/functions | entry_with_address.vdm |
| vdm-grouped/target/explicit/functions | get_entity_only_entry.vdm |
| vdm-grouped/target/explicit/functions | get_entry_with_ident.vdm |
| vdm-grouped/target/explicit/functions | get_entry.vdm |
| vdm-grouped/target/explicit/functions | identity_exists.vdm |
| vdm-grouped/target/explicit/functions | map_to_set.vdm |
| vdm-grouped/target/explicit/functions | test_invariant.vdm |
| vdm-grouped/target/explicit/functions | test_modify_invariant.vdm |
| vdm-grouped/target/explicit/operations | Add_Entity.vdm |
| vdm-grouped/target/explicit/operations | add_entry.vdm |
| vdm-grouped/target/explicit/operations | add_true_predicate.vdm |
| vdm-grouped/target/explicit/operations | Break_Handler.vdm |
| vdm-grouped/target/explicit/operations | Clear_Break.vdm |
| vdm-grouped/target/explicit/operations | clear_entry.vdm |
| vdm-grouped/target/explicit/operations | Continue_Program.vdm |
| vdm-grouped/target/explicit/operations | determine_break.vdm |
| vdm-grouped/target/explicit/operations | do_find_entity.vdm |
| vdm-grouped/target/explicit/operations | Establish_Finalisation.vdm |
| vdm-grouped/target/explicit/operations | establish_terminating.vdm |
| vdm-grouped/target/explicit/operations | Find_Entity.vdm |
| vdm-grouped/target/explicit/operations | get_entity.vdm |
| vdm-grouped/target/explicit/operations | get_entity_entry.vdm |
| vdm-grouped/target/explicit/operations | get_id_entry.vdm |
| vdm-grouped/target/explicit/operations | global_break_handler.vdm |
| vdm-grouped/target/explicit/operations | group_break_handler.vdm |
| vdm-grouped/target/explicit/operations | Initialise_Program.vdm |
| vdm-grouped/target/explicit/operations | local_break_handler.vdm |
| vdm-grouped/target/explicit/operations | local_break_response.vdm |
| vdm-grouped/target/explicit/operations | Modify_Break.vdm |
| vdm-grouped/target/explicit/operations | modify_entry_set.vdm |
| vdm-grouped/target/explicit/operations | modify_entry.vdm |
| vdm-grouped/target/explicit/operations | Predicates.vdm |
| vdm-grouped/target/explicit/operations | Program_Under_Test.vdm |
| vdm-grouped/target/explicit/operations | push_break_handler_msg.vdm |
| vdm-grouped/target/explicit/operations | push_saved_id.vdm |
| vdm-grouped/target/explicit/operations | Remove_Entity.vdm |
| vdm-grouped/target/explicit/operations | reset_break_event.vdm |
| vdm-grouped/target/explicit/operations | remove_delayed.vdm |
| vdm-grouped/target/explicit/operations | remove_entity_break_entries.vdm |
| vdm-grouped/target/explicit/operations | restore_entry.vdm |
| vdm-grouped/target/explicit/operations | restore_target_entry.vdm |
| vdm-grouped/target/explicit/operations | Set_Break.vdm |
| vdm-grouped/target/explicit/operations | set_break_entry.vdm |
| vdm-grouped/target/explicit/operations | set_break_event.vdm |
| vdm-grouped/target/explicit/operations | set_delayed_break.vdm |
| vdm-grouped/target/explicit/operations | set_entry.vdm |
| vdm-grouped/target/explicit/operations | set_finalisation.vdm |
| vdm-grouped/target/explicit/operations | set_termination.vdm |
| vdm-grouped/target/explicit/operations | update_modified_entry.vdm |
| vdm-grouped/target/explicit/operations | update_new_entry.vdm |
| vdm-grouped/target/explicit/operations | update_added_entry.vdm |

# Annex E

# External Namespace

### A Description of Contents

The contents of this annex represent the operational decompositions of the handlers of External namespace, the namespace which is responsible for maintaining a session context between debug agent and monitor server, the sending and receiving of messages and the management of communication channels.

| Pathname | Filename |
|---|---|
| vdm-grouped/external/explicit/operations | Close_Session.vdm |
| vdm-grouped/external/explicit/operations | Close_Read.vdm |
| vdm-grouped/external/explicit/operations | Close_Write.vdm |
| vdm-grouped/external/explicit/operations | do_cw.vdm |
| vdm-grouped/external/explicit/operations | do_close_write.vdm |
| vdm-grouped/external/explicit/operations | do_cr.vdm |
| vdm-grouped/external/explicit/operations | do_close_read.vdm |
| vdm-grouped/external/explicit/operations | do_open_read.vdm |
| vdm-grouped/external/explicit/operations | do_open_write.vdm |
| vdm-grouped/external/explicit/operations | Open_Session.vdm |
| vdm-grouped/external/explicit/operations | Open_Read.vdm |
| vdm-grouped/external/explicit/operations | Open_Write.vdm |
| vdm-grouped/external/explicit/operations | Receive.vdm |
| vdm-grouped/external/explicit/operations | Send.vdm |
| vdm-grouped/external/explicit/operations | Sendto.vdm |

# Annex F

# Modelling Data Communications

This annex on message reification is presented in printed form as an example of the steps taken *toward* a full refinement of a component of the specification. What is presented here is one of possibly many ways to data reify the message specification. The solution presented here is modelled after an ISO 8802-3 packet specification. The machine readable form of these specifications can be found in the directories *message/explicit/operations* (for operations) and *message* (for value and type definitions).

### BNF Message Grammar

*Request* =
*External-Name-Msg* | *Name-Msg* | *External-Break-Msg* | *User-Data-Msg*

*External-Name-Msg* =
*External-Name-Path* *Name-Path* Argument

*External-Name-Path* =
External-Name

*Name-Msg* =
*Name-Path* Argument

*Name-Path* =
$\text{Name}^{+}$

*External-Break-Msg* =
External-Break

*User-Data-Msg* =
User-Data

Annotate

External namespace is the only namespace that responds to external name tags. Other environments subordinate to it are just normal name environments.

End Annotation

*Reply* =
*Response-Msg* | *Data-Reponse-Msg* | *Error-Msg* | *Data-Error-Msg* | *User-Data* | *External-Break*

*Response-Msg* =
Response

*Data-Response-Msg* =
Response-Data

*Error-Msg* =
Error

*Data-Error-Msg* =
Error-Data

*User-Data* =
User-Data

*External-Break* =
External-Break

## Transmission Packets

types

*Binary-Digit* =
$\mathbf{N}$
inv *num* $\triangleq (num = 0) \vee (num = 1);$

*Bit-Sequence* =
$Binary\text{-}Digit^{+};$

*Octet* =
*Bit-Sequence*
inv *seq* $\triangleq$ len $seq = 8;$

*Tag* =
$Octet^{+}$
inv *seq* $\triangleq$
len $seq = 2;$

*Octet-Count* =
$\mathbf{N}$
inv *cnt* $\triangleq$
$(cnt > 0) \wedge (cnt \leq 2047);$

*Spare-Bits* =
$\mathbf{N}$
inv *cnt* $\triangleq$
$(cnt \geq 1) \wedge (cnt \leq 7);$

*Type-Field* =
*Bit-Sequence*
inv *seq* $\triangleq$ len $seq = 4;$

*Size-Field* =
*Bit-Sequence*
inv *seq* $\triangleq$ len $seq = 11;$

*Tag-Bits* ::
*type* : *Type-Field*
*size* : *Size-Field*
*spare* : *Binary-Digit*

*Spare* =
*Octet;*

*Spare-Data* =
*Spare-Bits;*

## Machine Independent Tag Values

values

*Name-Type-Size* : **N** =
*6;*

*Machine-Address-Type-Size* : **N** =
*6;*

*Address-Type-Size* : **N** =
*8;*

*Workspace-Type-Size* : **N** =
*8;*

*Break-Address-Type-Size* : **N** =
*8;*

*Count-Type-Size* : **N** =
*4;*

*Break-Kind-Type-Size* : **N** =
*1;*

*Break-Identity-Type-Size* : **N** =
*5;*

*Error-Type-Size* : **N** =
*1;*

*Response-Type-Size* : **N** =
*1;*

*Raw-Packet-Header-Size* : **N** =
*3;*

*Prefix-Size* : **N** =
*5;*

types

*conc-token* =
*Octet;*

*conc-Name* =
$Octet^+$
inv *seq* $\triangleq$ len *seq* = *Name-Type-Size;*

*conc-Address* =
$Octet^+$
inv *seq* $\triangleq$ len *seq* = *Address-Type-Size;*

*conc-External-Name* =
$Octet^+$
inv *seq* $\triangleq$ len *seq* = *Name-Type-Size;*

*conc-Machine-Address* =
$Octet^+$
inv *seq* $\triangleq$ len *seq* = *Machine-Addr-Type-Size;*

*conc-Predicate-Data* =
$Octet^+$
inv *seq* $\triangleq$ len *seq* = *Break-Address-Size;*

*conc-Break-Address* =
*conc-Address;*

*conc-Count* =
*Octet*$^+$
inv *seq* $\triangleq$ len *seq* = *Count-Type-Size;*

*conc-Workspace-Size* =
*Octet*$^+$
inv *seq* $\triangleq$ len *seq* = *Workspace-Type-Size;*

*conc-Error* =
*Octet*
inv *seq* $\triangleq$ len *seq* = *Error-Type-Size;*

*conc-Response* =
*Octet*
inv *seq* $\triangleq$ len *seq* = *Response-Type-Size*

*conc-Break-Identity* =
*Octet*$^+$
inv *b-id* $\triangleq$ len *b-id* = *Break-Identity-Type-Size;*

*conc-Break-Kind* =
*Octet*$^+$
inv *b-kind* $\triangleq$ len *b-kind* = *Break-Kind-Type-Size;*

## Model Handler Argument Concretions

*conc-Add-Name-Args* ::
*name : conc-Name*
*size : conc-Workspace-Size;*

*conc-Remove-Name-Args* ::
*name : conc-Name;*

*conc-Add-Handler-Args* ::
*name : conc-Name*
*size : conc-Workspace-Size*
*code : Octet*$^+$*;*

*conc-Remove-Handler-Args* ::
*name : conc-Name*

*conc-Replace-Handler-Args* ::
*name : conc-Name*
*size : conc-Workspace-Size*
*code : Octet*$^+$*;*

*conc-Set-Break-Args* ::
*break-kind : conc-Break-Kind*
*break-id : conc-Break-Id*
*break-addr : conc-Break-Address*
*break-entity : conc-Name;*

*conc-Modify-Break-Args* ::
*break-kind : conc-Break-Kind*
*break-id : conc-Break-Id*
*break-addr-old : conc-Break-Address*
*break-addr-new : conc-Break-Address*
*break-count : conc-Count*
*break-predicate : conc-Address*
*break-entity : conc-Name*
*break-netaddr : conc-Machine-Address;*

*conc-Clear-Break-Args* ::
  *break-kind* : *conc-Break-Kind*
  *break-id* : *conc-Break-Id*
  *break-addr* : *conc-Break-Address*;

*conc-Init-Program-Args* ::
  *put-pc* : *conc-Address*
  *put-sp* : *conc-Address*
  *put-ws-base* : *conc-Address*
  *put-ws-size* : *conc-Workspace-Size*;

*conc-Set-Finalisation-Args* ::
  *put-final-pc* : *conc-Address*;

*conc-Add-Hardware-Args* ::
  *name* : *conc-Name*
  *port-cnt* : *conc-Count*
  *hw-descr* : $Octet^+$;

*conc-Open-Read-Args* ::
  *name* : *conc-Name*;

*conc-Close-Read-Args* ::
  *name* : *conc-Name*;

*conc-Open-Write-Args* ::
  *name* : *conc-Name*;

*conc-Close-Write-Args* ::
  *name* : *conc-Name*;

*conc-Check-Driver-Args* ::
  *name* : *conc-Name*
  *port-cnt* : *conc-Count*
  *hw-descr* : $Octet^+$;

*conc-Init-Driver-Args* ::
  *name* : *conc-Name*
  *port-cnt* : *conc-Count*
  *hw-descr* : $Octet^+$;

*conc-Model-Handler-Args* =
  *conc-Add-Name-Args* | *conc-Remove-Name-Args* |
  *conc-Add-Handler-Args* | *conc-Remove-Handler-Args* |
  *conc-Replace-Handler-Args* | *conc-Set-Break-Args* |
  *conc-Modify-Break-Args* | *conc-Clear-Break-Args* |
  *conc-Init-Program-Args* | *conc-Set-Finalisation-Args* |
  *conc-Add-Hardware-Args* | *conc-Open-Read-Args* |
  *conc-Open-Write-Args* | *conc-Close-Read-Args* |
  *conc-Close-Write-Args* | *conc-Init-Driver-Args* |
  *conc-Check-Driver-Args*;

*conc-Break-Message* ::
  *b-kind* : *conc-Break-Kind*
  *b-id* : *conc-Break-Identity*;

*conc-Message-Body* =
  $Octet^+$ |
  *conc-Name* |
  *conc-External-Name* |
  *conc-Break-Message* |
  *conc-Model-Handler-Args*;

```
conc-Msg-Element ::
                    tag : Tag-Bits
                    spare : [Spare]
                    value : conc-Message-Body
                       inv element  ≜
                            let mk-conc-Msg-Element(tg,_,val) = element in
                              let mk-Tag-Bits(_,size,_) = tg in
                                len val = retr-Nat(size);
```

## Concrete Message Format

```
types
   conc-Raw-Msg-Length =
                        Octet+
                           inv pk-len  ≜ len pk-len = 2;

   Raw-Packet ::
              destination : conc-Machine-Address
              source : conc-Machine-Address
              length : conc-Raw-Msg-Length
              message : conc-Raw-Packet
                 inv mk-conc-Raw-Packet(destination,source,length,raw-message)  ≜
                      let mk-conc-Raw-Packet(_,_,_,data) = raw-message in
                        let bseq : Bit-Sequence = octet-to-bits(length) in
                          (retr-Nat(bseq) = Raw-Packet-Header-Size + len data) ∧
                          (destination ≠ source);

   conc-Raw-Packet ::
                    dest-port : Octet
                    src-port : Octet
                    control : Octet
                    data : conc-Raw-Message;

   conc-Raw-Message =
                    Octet+;
```

## Auxiliary Functions to Map Concretions to Model Values

```
functions
bin-map : Binary-Digit → N
bin-map(bit)  ≜
1. (
2.    if bit = 1 then
3.       1
4.    else
5.       0
6. )

retr-Nat : Bit-Sequence → N
retr-Nat(bitseq)  ≜
1. (
2.    let head = hd bitseq,rest = tl bitseq in
3.       if head = [] then
4.          0
5.       else
6.          retr-Nat(rest) × 2 + bin-map(head)
7. )
```

```
retr-Count : Bit-Sequence → N
retr-Count(bitseq) ≜
1. (
2.    return retr-Nat(bitseq);
3. )

retr-Addr : Bit-Sequence → N
retr-Addr(bitseq) ≜
1. (
2.    return retr-Nat(bitseq);
3. )
```

The *Count* and *Address* types all use *retr-Nat* to take a concretion and convert this concretion into a model type.

## Message Retrievals

```
functions
retrieve-Message : conc-Raw-Message → Message
retrieve-Message(raw-msg) ≜
1. (
2.    let mk-(msg,msg-ok) = retr-Message(raw-msg,
3.                                  mk-Msg-Element(⟨NAME⟩, ⟨ROOT⟩)) in
4.      if msg-ok then
5.        msg
6.      else
7.        mk-Message(mk-Msg-Element(⟨ERROR⟩,⟨BAD-MESSAGE-FORMAT⟩))
8. )

retr-Message : conc-Raw-Message × Msg-Element → [Message] × B
retr-Message(raw-msg,previous) ≜
1. (
2.    if len raw-msg = 0 then
3.      mk-(nil,true)
4.    else
5.      let mk-(length,element) = retr-Msg-Element(octets,prev) in
6.        if element = nil then
7.          mk-(nil,false)
8.        else
9.          let used ⌢ rest = raw-msg in
10.            let used ⌢ rest : conc-Raw-Message be st len used = length in
11.              let mk-(msg,result) = retr-Message(rest,element) in
12.                if result then
13.                  mk-(msg ⌢ [element],result)
14.                else
15.                  mk-(nil,false)
16. )

retr-Msg-Element : conc-Raw-Message × Msg-Element → N × [Msg-Element]
retr-Msg-Element(raw-msg,prev) ≜
1. (
2.    if len raw-msg < 3 then
3.      mk-(len raw-msg,nil)
4.    else
5.      let rest ⌢ tag = raw-msg in
6.        let rest ⌢ tag : conc-Raw-Message be st len tag = 2 in
7.          let mk-(length,kind,pad,err) = unhatch-tag(tag) in
8.            if err then
9.              mk-(length,nil)
10.           else
```

```
11.              if length > len rest then
12.                 mk-(length,nil)
13.              else
14.                 if pad then
15.                    let padbits ⌢ value ⌢ tail = rest in
16.                       let padbits ⌢ value ⌢ tail : conc-Raw-Message be st
17.                          (len padbits = 1) ∧ (len value = length) in
18.                          let padcnt : N = retr-Nat(octet-to-bits(padbits)) in
19.                             let val ⌢ padding = value in
20.                                let val ⌢ padding : conc-Raw-Message = value be st
21.                                   len padding = padcnt in
22.                                   mk-(length + 3,rx-filter(prev,length,kind,value))
23.                 else
24.                    let rest = value ⌢ tail in
25.                       let value ⌢ tail : conc-Raw-Message be st len value = length in
26.                          mk-(length + 2,rx-filter(prev,length,kind,value))
27. )
```

Annotate

8-9, 11-12. If a tag or length is bad then the message is bad.

14-22. This removes any octet padding that may have been required in a message element whose contents may not have amounted to an integral number of octets.

23-26. The message element is an integral number of octets in size.

End Annotation

The *unhatch-tag* function takes a tag and extracts any tag information required by the retrieval, the *Message-Kind*, value length and padding indicator. The message kind is determined by using *get-tag* which performs a bit sequence pattern match to yield the kind.

```
functions
unhatch-tag : Octet+ → N × Message-Kind × B × B
unhatch-tag(octets) ≜
1. (
2.    let tag : Tag-Bits be st octets = tagbits-to-octets(tag) in
3.       let mk-Tag-Bits(type,size,sp) = tag in
4.          let mk-(kind,error) = get-tag(type),
5.              sz : N = retr-Nat(size),
6.              spare : B = (sp = 1) in
7.             mk-(kind,sz,spare,error)
8. )

get-tag : Type-Field → Message-Kind × B
get-tag(tg) ≜
1. (
2.    cases tg:
3.    [0,0,0,0] → mk-(⟨NAME⟩,false),
4.    [0,0,0,1] → mk-(⟨EXTERNAL-NAME⟩,false),
5.    [0,0,1,0] → mk-(⟨EXTERNAL-BREAK⟩,false),
6.    [0,0,1,1] → mk-(⟨USER-DATA⟩,false),
7.    [0,1,0,0] → mk-(⟨RESPONSE⟩,false),
8.    [0,1,0,1] → mk-(⟨ARGUMENT⟩,false),
9.    [0,1,1,0] → mk-(⟨ERROR⟩,false),
10.    [0,1,1,1] → mk-(⟨DATA-RESPONSE⟩,false),
11.    [1,0,0,0] → mk-(⟨DATA-ERROR⟩,false),
12.    others → mk-(⟨ERROR⟩,true)
13.    end
14. )
```

```
rx-filter : Msg-Element × N × Message-Kind × conc-Message-Body → [Msg-Element]
rx-filter(prev,length,kind,raw-value) ≜
1.  (
2.    let mk-Msg-Element(prev-kind,prev-value) = prev in
3.    cases kind:
4.    ⟨NAME⟩ → Check-Name(length,prev-kind,kind,raw-value),
5.    ⟨EXTERNAL-NAME⟩ → Check-External-Name(length,prev-kind,kind,raw-value),
6.    ⟨EXTERNAL-BREAK⟩ → Check-External-Break(length,prev-kind,kind,raw-value),
7.    ⟨USER-DATA⟩ →
8.      cases prev-kind:
9.      ⟨NAME⟩ →
10.        if prev-value = ⟨ROOT⟩ then
11.          let value : token+ = retr-tokens(raw-value) in
12.            mk-Msg-Element(kind,value)
13.        else
14.          nil,
15.      ⟨EXTERNAL-NAME⟩,⟨EXTERNAL-BREAK⟩,
16.      ⟨USER-DATA⟩,⟨ARGUMENT⟩ → nil
17.      end,
18.    ⟨RESPONSE⟩,⟨RESPONSE-DATA⟩,
19.    ⟨ERROR⟩,⟨ERROR-DATA⟩ → nil,
20.    ⟨ARGUMENT⟩ →
21.      cases prev-kind:
22.      ⟨NAME⟩,⟨EXTERNAL-NAME⟩ →
23.        Check-Arguments(length,prev-value,raw-value),
24.      ⟨EXTERNAL-BREAK⟩,⟨USER-DATA⟩,⟨ARGUMENT⟩ →
25.        nil
26.      end
27.    end
28. )
```

Annotate

18-19. The monitor cannot receive responses or error notifications from the debug agent.

21-26. There are no case guards for errors and responses because they are caught within the body of the main case and therefore need not be within the ARGUMENT case statement.

End Annotation

The following functions determine whether a message element is of a particular kind and construct the appropriate *Msg-Element*. The *retr-Name* operation returns the name denoted by that bit pattern.

```
types
  Name-Map =
          Octet -m-> Name;
values
  Null-Prefix : Octet+ =
          [[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0]];
  decode-map : Name-Map =
          {
          [0,0,0,0,0,0,0,0] ↦ ⟨SELF⟩,
          [0,0,0,0,0,0,0,1] ↦ ⟨ADD-NAME⟩,
          [0,0,0,0,0,0,1,0] ↦ ⟨REMOVE-NAME⟩,
          [0,0,0,0,0,0,1,1] ↦ ⟨ADD-HANDLER⟩,
          [0,0,0,0,0,1,0,0] ↦ ⟨REMOVE-HANDLER⟩,
          [0,0,0,0,0,1,0,1] ↦ ⟨REPLACE-HANDLER⟩,
          [0,0,0,0,0,1,1,0] ↦ ⟨ADD-ENTITY⟩,
```

```
[0,0,0,0,0,1,1,1] ↦ ⟨REMOVE-ENTITY⟩,
[0,0,0,0,1,0,0,0] ↦ ⟨FIND-ENTITY⟩,
[0,0,0,0,1,0,0,1] ↦ ⟨SET-BREAK⟩,
[0,0,0,0,1,0,1,0] ↦ ⟨MODIFY-BREAK⟩,
[0,0,0,0,1,0,1,1] ↦ ⟨CLEAR-BREAK⟩,
[0,0,0,0,1,1,0,1] ↦ ⟨HANDLE-BREAK⟩,
[0,0,0,0,1,1,0,1] ↦ ⟨INIT-PROG⟩,
[0,0,0,0,1,1,1,0] ↦ ⟨SET-FINALISATION⟩,
[0,0,0,0,1,1,1,1] ↦ ⟨CONT-PROG⟩,
[0,0,0,1,0,0,0,0] ↦ ⟨ADD-HARDWARE⟩,
[0,0,0,1,0,0,0,1] ↦ ⟨OPEN-READ⟩,
[0,0,0,1,0,0,1,0] ↦ ⟨OPEN-WRITE⟩,
[0,0,0,1,0,0,1,1] ↦ ⟨CLOSE-READ⟩,
[0,0,0,1,0,1,0,0] ↦ ⟨CLOSE-WRITE⟩,
[0,0,0,1,0,1,0,1] ↦ ⟨SEND⟩,
[0,0,0,1,0,1,1,0] ↦ ⟨SENDTO⟩,
[0,0,0,1,0,1,1,1] ↦ ⟨RECEIVE⟩,
[0,0,0,1,1,0,0,0] ↦ ⟨CHECK-DRIVER⟩,
[0,0,0,1,1,0,0,1] ↦ ⟨OPEN-SESSION⟩,
[0,0,0,1,1,0,1,0] ↦ ⟨CLOSE-SESSION⟩,
[0,0,0,1,1,0,1,1] ↦ ⟨INITIALISE⟩,
[0,0,0,1,1,1,0,0] ↦ ⟨ROOT⟩,
[0,0,0,1,1,1,0,1] ↦ ⟨EXTERNAL⟩,
[0,0,0,1,1,1,1,0] ↦ ⟨DEVICE⟩,
[0,0,0,1,1,1,1,1] ↦ ⟨TARGET⟩,
} ⊎
{key ↦ ⟨UNKNOWN-MODEL-NAME⟩ | key : Octet · (retr-
Nat(key) >= 32 and retr-Nat(key) <= 255)};

functions
retr-token : conc-token → token
retr-token(raw-tok) ≜
1. (
2.    mk-token(raw-tok)
3. )

retr-tokens : conc-token+ → token+
retr-tokens(raw-toks) ≜
1. (
2.    let head = hd raw-toks,rest = tl raw-toks in
3.       if rest = [] then
4.          [retr-token(head)]
5.       else
6.          retr-token(head) ⌢ retr-tokens(rest)
7. )

to-char : Octet → char
to-char(oct-ch) ≜
1. (
2.    let res : char be st mk-token(res) = retr-token(oct-ch) in
3.       res
4. )

retr-String : Octet+ → char+
retr-String(oct-str) ≜
1. (
2.    let s1 = hd oct-str,s2 = tl oct-str in
3.       if s2 = [] then
4.          [to-char(s1)]
5.       else
6.          to-chars(s1) ⌢ retr-String(s2)
7. )
```

```
retr-Name : conc-Name → Name
retr-Name(raw-name)  ≜
1.  (
2.     let prefix ⌢ name = raw-name in
3.       let prefix ⌢ name : Octet⁺ be st len prefix = Prefix-Size in
4.         if prefix = Null-Prefix then
5.           decode-map(name(1))
6.       else
7.         let name : Name be st name = mk-token(retr-tokens(raw-name)) in
8.           name
9.  )
```

```
retr-breakkind : conc-Break-Kind → Break-Kind
retr-breakkind(raw-break-kind)  ≜
1.  (
2.     if raw-break-kind = [0,0,0,0,0,0,0,0] then
3.       ⟨LOCAL⟩
4.     else if raw-break-kind = [0,0,0,0,0,0,0,1] then
5.       ⟨GROUP⟩
6.     else if raw-break-kind = [0,0,0,0,0,0,1,0] then
7.       ⟨DELAYED⟩
8.     else if raw-break-kind = [0,0,0,0,0,0,1,1] then
9.       ⟨GLOBAL⟩
10.    else
11.      ⟨LOCAL⟩
12. )
```

```
retr-Port : Octet⁺ → Port⁺
retr-Port(port-stream)  ≜
1.  (
2.     let base ⌢ size ⌢ rest = port-stream in
3.       let base ⌢ size ⌢ rest : Octet⁺ be st
4.           (len base = Address-Type-Size) ∧ (len size = Count-Type-Size) in
5.         if rest = [] then
6.           [mk-Port(retr-Addr(base),retr-Nat(size))]
7.         else
8.           mk-Port({retr-Addr(octet-to-bits(base)) ↦ undefined},
9.                     retr-Nat(octet-to-bits(size))) ⌢ retr-Port(rest)
10. )
```

```
retr-Hardware-Resource : N × Octet⁺ → Hardware-Resource
retr-Hardware-Resource(count,descr-stream)  ≜
1.  (
2.     let ports ⌢ prod-id ⌢ man-id ⌢ ser-id ⌢ comment = descr-stream in
3.       let ports ⌢ prodid ⌢ man-id ⌢ ser-id ⌢ comment : Octet⁺ be st
4.           (len ports = count × (Address-Type-Size + Count-Type-Size)) ∧
5.           (len prod-id = Count-Type-Size) ∧
6.           (len man-id = Count-Type-Size) ∧
7.           (len ser-id = Address-Type-Size) in
8.         let comment-length ⌢ comment-content = comment in
9.           let comment-length ⌢ comment-content : Octet⁺ be st
10.              (len comment-length = Count-Type-Size) in
11.            let length : N = retr-Nat(octet-to-bits(comment-length)) in
12.              let port-descr : Port⁺ = retr-Port(ports),
13.                  product : token⁺ = mk-token(retr-tokens(prod-id)),
14.                  manufacturer : token⁺ = mk-token(retr-tokens(man-id)),
15.                  serial : token⁺ = mk-token(retr-tokens(ser-id)),
16.                  comment-string : char⁺ = retr-String(comment-content) in
17.                mk-Hardware-Resource(port-descr,product,manufacturer,
18.                                     serial,comment-string)
19. )
```

```
Check-Name : N × Message-Kind × Message-Kind × conc-Message-Body → [Msg-Element]
Check-Name(length,prev-kind,kind,raw-value) ≜
1.  (
2.     if length = Name-Type-Size then
3.     (
4.        let name-val = retr-Name(raw-value) in
5.           if prev-kind ∈ {⟨ARGUMENT⟩,⟨EXTERNAL-BREAK⟩,⟨USER-DATA⟩} then
6.              nil
7.           else
8.              mk-Msg-Element(kind,name-val)
9.     )
10.    else
11.       nil
12. )

Check-External-Name : N × Msg-Element ×
                      Message-Kind × conc-Message-Body → [Msg-Element]
Check-External-Name(length,previous,kind,raw-value) ≜
1.  (
2.     if length = Name-Type-Size then
3.     (
4.        let name-val = retr-Name(raw-value) in
5.           let mk-Msg-Element(prev-kind,prev-value) = previous in
6.              cases prev-kind:
7.              ⟨NAME⟩ →
8.                 if prev-value = ⟨ROOT⟩ then
9.                    mk-Msg-Element(kind,name-val)
10.                else
11.                   nil,
12.             ⟨ARGUMENT⟩,⟨EXTERNAL-BREAK⟩,⟨USER-DATA⟩ →
13.                nil
14.             end
15.    )
16.    else
17.       nil
18. )

Check-External-Break : N × Msg-Element ×
Message-Kind × conc-Message-Body → [Msg-Element]
Check-External-Break(length,previous,kind,raw-value) ≜
1.  (
2.     if length = Break-Kind-Type-Size + Break-Identity-Type-Size then
3.        let mk-Msg-Element(prev-kind,prev-value) = previous in
4.           let b-kind ⌢ b-id = raw-value in
5.              let b-kind : conc-Break-Kind = raw-value(0) in
6.                 let b-id : Octet+ be st
7.                    (len b-id = Break-Identity-Type-Size) in
8.                    let kind : Breakpoint-Kind = retr-breakkind(b-kind),
9.                       id : token = mk-token(retr-tokens(b-id)) in
10.                        cases prev-kind:
11.                        ⟨NAME⟩ →
12.                           if prev-value = ⟨ROOT⟩ then
13.                              let body = mk-Break-Identity(kind,id) in
14.                                 mk-Msg-Element(⟨EXTERNAL-BREAK⟩,body)
15.                           else
16.                              nil,
17.                        ⟨EXTERNAL-NAME⟩,⟨EXTERNAL-BREAK⟩,
18.                        ⟨USER-DATA⟩,⟨ARGUMENT⟩ →
19.                           nil
```

```
20.                 end
21.     else
22.        nil
23.  )

Check-Arguments : N × Name × conc-Message-Body → [Msg-Element]
Check-Arguments(pack-length,prev-name,pack) ≜
1.  (
2.     cases prev-name:
3.     ⟨ADD-NAME⟩ →
4.        if pack-length = Name-Type-Size + Workspace-Type-Size then
5.           mk-Msg-Element(⟨ARGUMENT⟩,retr-Add-Name-Args(pack))
6.        else
7.           nil,
8.     ⟨REMOVE-NAME⟩ →
9.        if pack-length = Name-Type-Size then
10.           mk-Msg-Element(⟨ARGUMENT⟩,retr-Remove-Name-Args(pack))
11.       else
12.          nil,
13.    ⟨ADD-HANDLER⟩ →
14.       let name ⌢ size ⌢ code = pack in
15.          let name ⌢ size ⌢ code : conc-Message-Body be st
16.              (len name = Name-Type-Size) ∧
17.              (len size = Workspace-Type-Size) in
18.            mk-Msg-Element(⟨ARGUMENT⟩,retr-Add-Handler-Args(pack)),
19.    ⟨REMOVE-HANDLER⟩ →
20.       if pack-length = Name-Type-Size then
21.          mk-Msg-Element(⟨ARGUMENT⟩,retr-Remove-Handler-Args(pack))
22.       else
23.          nil,
24.    ⟨REPLACE-HANDLER⟩ →
25.       let name ⌢ size ⌢ code = pack in
26.          let name ⌢ size ⌢ code : conc-Message-Body be st
27.              (len name = Name-Type-Size) ∧
28.              (len size = Workspace-Type-Size) in
29.            mk-Msg-Element(⟨ARGUMENT⟩,retr-Replace-Handler-Args(pack)),
30.    ⟨ADD-ENTITY⟩,⟨REMOVE-ENTITY⟩,⟨FIND-ENTITY⟩ →
31.       nil,
32.    ⟨SET-BREAK⟩ →
33.       if pack-length = Break-Identity-Type-Size + Break-Kind-Type-Size +
34.                        Address-Type-Size + Name-Type-Size then
35.          mk-Msg-Element(⟨ARGUMENT⟩,retr-Set-Break-Args(pack))
36.       else
37.          nil,
38.    ⟨MODIFY-BREAK⟩ →
39.       if pack-length = Break-Identity-Type-Size + Break-Kind-Type-Size +
40.                        Address-Type-Size + Address-Type-Size + Count-Type-Size +
41.                        Address-Type-Size + Name-Type-Size +
42.                        Machine-Address-Type-Size then
43.          mk-Msg-Element(⟨ARGUMENT⟩,retr-Modify-Break-Args(pack))
44.       else
45.          nil,
46.    ⟨CLEAR-BREAK⟩ →
47.       if pack-length = Break-Identity-Type-Size + Break-Kind-Type-Size +
48.                        Address-Type-Size then
49.          mk-Msg-Element(⟨ARGUMENT⟩,retr-Clear-Break-Args(pack))
50.       else
51.          nil,
52.    ⟨HANDLE-BREAK⟩ →
53.       nil,
```

```
54.   ⟨INIT-PROG⟩ →
55.     if pack-length = Address-Type-Size + Address-Type-Size +
56.                      Address-Type-Size + Workspace-Type-Size then
57.       mk-Msg-Element(⟨ARGUMENT⟩,retr-Initialise-Program-Args(pack))
58.     else
59.       nil,
60.   ⟨SET-FINALISATION⟩ →
61.     if pack-length = Address-Type-Size then
62.       mk-Msg-Element(⟨ARGUMENT⟩,retr-Establish-Finalisation-Args(pack))
63.     else
64.       nil,
65.   ⟨CONT-PROG⟩ →
66.     nil,
67.   ⟨ADD-HARDWARE⟩ →
68.     let name ⌢ port-count ⌢ hw-descr = pack in
69.       let name ⌢ port-count ⌢ hw-descr : conc-Message-Body be st
70.           (len name = Name-Type-Size) ∧ (len port-count = Count-Type-Size) in
71.         let count : N = retr-Nat(octet-to-bits(port-count)) in
72.           let dev-name : Name = retr-Name(name) in
73.             let hw-resource : Hardware-Resource = retr-Hardware-Resource(count,hw-
descr) in
74.               mk-Msg-Element(⟨ARGUMENT⟩,
75.                              mk-Add-Hardware-Args(dev-name,hw-resource,nil)),
76.   ⟨OPEN-READ⟩ →
77.     if pack-length = Name-Type-Size then
78.       mk-Msg-Element(⟨ARGUMENT⟩,retr-Open-Read-Args(pack))
79.     else
80.       nil,
81.   ⟨OPEN-WRITE⟩ →
82.     if pack-length = Name-Type-Size then
83.       mk-Msg-Element(⟨ARGUMENT⟩,retr-Open-Write-Args(pack))
84.     else
85.       nil,
86.   ⟨CLOSE-READ⟩ →
87.     if pack-length = Name-Type-Size then
88.       mk-Msg-Element(⟨ARGUMENT⟩,retr-Close-Read-Args(pack))
89.     else
90.       nil,
91.   ⟨CLOSE-WRITE⟩ →
92.     if pack-length = Name-Type-Size then
93.       mk-Msg-Element(⟨ARGUMENT⟩,retr-Close-Write-Args(pack))
94.     else
95.       nil,
96.   ⟨SEND⟩ → nil,
97.   ⟨SENDTO⟩ → nil,
98.   ⟨RECEIVE⟩ → nil,
99.   ⟨CHECK-DRIVER⟩ →
100.    let name ⌢ port-count ⌢ hw-descr = pack in
101.      let name ⌢ port-count ⌢ hw-descr : conc-Message-Body be st
102.          (len name = Name-Type-Size) ∧ (len port-count = Count-Type-Size) in
103.        let count : N = retr-Nat(octet-to-bits(port-count)) in
104.          let dev-name : Name = retr-Name(name) in
105.            let hw-resource : Hardware-Resource = retr-Hardware-Resource(count,hw-
descr) in
106.              let mk-Hardware-Resource(portlist,_,_,_,_) = hw-resource in
107.              mk-Msg-Element(⟨ARGUMENT⟩,
108.                             mk-Check-Driver-Args(dev-name,portlist)),
109.  ⟨OPEN-SESSION⟩ → nil,
110.  ⟨CLOSE-SESSION⟩ → nil,
```

```
111.    ⟨INITIALISE⟩ →
112.       let name ⌢ port-count ⌢ hw-descr = pack in
113.          let name ⌢ port-count ⌢ hw-descr : conc-Message-Body be st
114.               (len name = Name-Type-Size) ∧ (len port-count = Count-Type-Size) in
115.             let count : N = retr-Nat(octet-to-bits(port-count)) in
116.                let dev-name : Name = retr-Name(name) in
117.                 let hw-resource : Hardware-Resource = retr-Hardware-Resource(count,hw-
descr) in
118.                   let mk-Hardware-Resource(portlist,_,_,_,_) = hw-resource in
119.                      mk-Msg-Element(⟨ARGUMENT⟩,
120.                                              mk-Init-Driver-Args(dev-name,portlist)),
121.    others →
122.       mk-Msg-Element(⟨ARGUMENT⟩,retr-tokens(pack))
123.    end
124. )
```

Annotate

13-18, 24-29. Other than the upper bound imposed by the length of the message element there is no fixed pack length for the Add and Replace handler functions.

30-31. The entity operations are only invoked by the monitor and the PUT. The agent cannot use these handlers explicitly.

52-53. The break event handler cannot be explicitly invoked by the agent.

65-66. If the message is well-formed then control should not reach here due to the fact that starting or resuming the PUT does not require arguments. It is an error if an argument pack exists for this handler.

96-98. The agent cannot explicitly invoke any input-output handler.

121-122. User defined handlers for a given namespace have no conversion done.

End Annotation

```
functions
retr-Add-Name-Args : conc-Message-Body → Message-Body
retr-Add-Name-Args(raw-pack)  ≜
1. (
2.    let mk-conc-Add-Name-Args(cname,csize) = raw-pack in
3.       let name : Name = retr-Name(cname),
4.            size : N1 = retr-Nat(octet-to-bits(csize)) in
5.          mk-Add-Name-Args(name,size)
6. )

retr-Remove-Name-Args : conc-Message-Body → Message-Body
retr-Remove-Name-Args(raw-pack)  ≜
1. (
2.    let mk-conc-Remove-Name-Args(cname) = raw-pack in
3.       let name : Name = retr-Name(cname) in
4.          mk-Remove-Name-Args(name)
5. )

retr-Add-Handler-Args : conc-Message-Body → Message-Body
retr-Add-Handler-Args(raw-pack)  ≜
1. (
2.    let mk-conc-Add-Handler-Args(cname,csize,ccode) = raw-pack in
3.       let name : Name = retr-Name(cname),
4.            size : N1 = retr-Nat(octet-to-bits(csize)),
5.            code : token+ = retr-tokens(ccode) in
6.          mk-Add-Handler-Args(name,size,code)
7. )
```

```
retr-Remove-Handler-Args : conc-Message-Body → Message-Body
retr-Remove-Handler-Args(raw-pack)  ≜
1.  (
2.     let mk-conc-Remove-Handler-Args(cname) = raw-pack in
3.        let name : Name = retr-Name(cname) in
4.           mk-Remove-Handler-Args(name)
5.  )
retr-Replace-Handler-Args : conc-Message-Body → Message-Body
retr-Replace-Handler-Args(raw-pack)  ≜
1.  (
2.     let mk-conc-Replace-Handler-Args(cname,csize,ccode) = raw-pack in
3.        let name : Name = retr-Name(cname),
4.              size : N_1 = retr-Nat(octet-to-bits(csize)),
5.              code : token^+ = retr-tokens(ccode) in
6.           mk-Replace-Handler-Args(name,size,code)
7.  )

retr-Set-Break-Args : conc-Message-Body → Message-Body
retr-Set-Break-Args(raw-pack)  ≜
1.  (
2.     let mk-conc-Set-Break-Args(c-kind,c-id,c-addr,c-ename) = raw-pack in
3.        let kind : Breakpoint-Kind = decode-break(c-kind),
4.              id : token = mk-token(retr-tokens(c-id)),
5.              addr : Address = {retr-Addr(octet-to-bits(c-addr)) ↦ undefined},
6.              ename : Name = retr-Name(c-ename) in
7.           mk-Set-Break-Args(kind,id,addr,ename)
8.  )

retr-Modify-Break-Args : conc-Message-Body → Message-Body
retr-Modify-Break-Args(raw-pack)  ≜
1.  (
2.     let mk-conc-Modify-Break-Args(c-kind,c-id,c-old-addr,c-new-addr,
3.                                   c-new-cnt,c-new-pred-data,
4.                                   c-new-ename,c-machine-addr) = raw-pack in
5.        let kind : Breakpoint-Kind = decode-break(c-kind),
6.              id : token = mk-token(retr-tokens(c-id)),
7.              old-addr : Address = {retr-Addr(octet-to-bits(c-old-addr)) ↦ undefined},
8.              new-addr : Address = {retr-Addr(octet-to-bits(c-new-addr)) ↦ undefined},
9.              new-cnt : N = retr-Count(c-new-cnt),
10.                   new-pred-data : Address = {retr-Addr(octet-to-bits(c-new-pred-data)) ↦
undefined},
11.             new-ename : Name = retr-Name(c-new-ename),
12.             new-machine-addr : Machine-Address = mk-token(retr-tokens(c-machine-addr))
in
13.           return mk-Modify-Break-Args(kind,id,
14.                                        old-addr,new-addr,
15.                                        new-cnt,new-pred-data,
16.                                        new-ename,new-machine-addr)
17.  )

retr-Clear-Break-Args : conc-Message-Body → Message-Body
retr-Clear-Break-Args(raw-pack)  ≜
1.  (
2.     let mk-conc-Clear-Break-Args(c-kind,c-id,c-addr) = raw-pack in
3.        let kind : Breakpoint-Kind = retr-breakkind(c-kind),
4.              id : token = mk-token(retr-tokens(c-id)),
5.              addr : Address = {retr-Name(octet-to-bits(c-addr)) ↦ undefined} in
6.           mk-Clear-Break-Args(kind,id,addr)
7.  )
```

*retr-Initialise-Program-Args : conc-Message-Body → Message-Body*
*retr-Initialise-Program-Args(raw-pack)* △
1. (
2. let *mk-conc-Initialise-Program-Args(c-initpc,c-initsp,c-wsbase,c-wssize) = raw-pack* in
3. let *initpc : Address = {retr-Addr(octet-to-bits(c-initpc))* ↦ undefined},
4. *initsp : Address = {retr-Addr(octet-to-bits(c-initsp))* ↦ undefined},
5. *wsbase : Address = {retr-Addr(octet-to-bits(c-wsbase))* ↦ undefined},
6. *wssize* : $\mathbf{N}_1$ = *retr-Nat(octet-to-bits(c-wssize))* in
7. *mk-Initialise-Program-Args(initpc,initsp,wsbase,wssize)*
8. )

*retr-Establish-Finalisation-Args : conc-Message-Body → Message-Body*
*retr-Establish-Finalisation-Args(raw-pack)* △
1. (
2. let *mk-conc-Establish-Finalisation-Args(c-finalisation) = raw-pack* in
3. let *final : Address = {retr-Addr(octet-to-bits(c-finalisation))* ↦ undefined} in
4. *mk-Establish-Finalisation-Args(final)*
5. )

*retr-Open-Read-Args : conc-Message-Body → Message-Body*
*retr-Open-Read-Args(raw-pack)* △
1. (
2. let *mk-conc-Open-Read-Args(cname) = raw-pack* in
3. let *driver-name : Name = retr-Name(cname)* in
4. *mk-Open-Read-Args(driver-name)*
5. )

*retr-Open-Write-Args : conc-Message-Body → Message-Body*
*retr-Open-Write-Args(raw-pack)* △
1. (
2. let *mk-conc-Open-Write-Args(cname,cagentaddress) = raw-pack* in
3. let *name : Name = retr-Name(cname),*
4. *agent-addr : Machine-Address = mk-*token*(retr-tokens(cagentaddress))* in
5. *mk-Open-Write-Args(name,agent-addr)*
6. )

*retr-Close-Read-Args : conc-Message-Body → Message-Body*
*retr-Close-Read-Args(raw-pack)* △
1. (
2. let *mk-conc-Close-Read-Args(cname) = raw-pack* in
3. let *driver-name : Name = retr-Name(cname)* in
4. *mk-Close-Read-Args(driver-name)*
5. )

*retr-Close-Write-Args : conc-Message-Body → Message-Body*
*retr-Close-Write-Args(raw-pack)* △
1. (
2. let *mk-conc-Close-Write-Args(cname) = raw-pack* in
3. let *driver-name : Name = retr-Name(cname)* in
4. *mk-Close-Write-Args(driver-name);*
5. )

## Auxiliary Functions to Map Model Values to Concretions

functions
*conc-Nat* : $\mathbf{N} \to Octet^+$
*conc-Nat(num)* △
1. (
2. let *raw-num* : $Octet^+$ be st *num = retr-Nat(octet-to-bits(raw-num))* in
3. *raw-num*
4. )

```
conc-Size-Field : N → Size-Field
conc-Size-Field(sz) ≜
1. (
2.    let raw-sz : Size-Field be st sz = retr-Nat(raw-sz) in
3.       raw-sz
4. )
```

## Message Concretion

```
functions
conc-Message : Message → [Octet+]
conc-Message(model-msg) ≜
1. (
2.    if len model-msg = 1 then
3.       conc-MsgElement(hd model-msg)
4.    else
5.       nil
6. )

conc-MsgElement : Msg-Element → [Octet+]
conc-MsgElement(model-element) ≜
1. (
2.    let mk-Msg-Element(kind,value) = model-element in
3.       let raw-value =
4.       (
5.          cases kind:
6.          ⟨ERROR⟩ →
7.             conc-error(value),
8.          ⟨RESPONSE⟩ →
9.             conc-response(value),
10.          ⟨USER-DATA⟩,⟨ERROR-DATA⟩,⟨RESPONSE-DATA⟩ →
11.             conc-data(value),
12.          ⟨EXTERNAL-BREAK⟩ →
13.             conc-break(value),
14.          others →
15.             nil
16.          end
17.       ) in
18.       let raw-value-bits : Bit-Sequence = raw-value in
19.          let extra = len raw-value-bits mod 8 in
20.             let pad : Spare-Bits = if extra > 0 then 8 − extra else 0 in
21.                let padding : Bit-Sequence be st len padding = pad in
22.                   let raw-tag = make-tag(kind),
23.                       size = len raw-value-octets,
24.                       spare = if extra = 0 then 0 else 1 in
25.                   let sz-fld : Size-Field = conc-Size-Field(size),
26.                       sp-fld : Binary-Digit = if spare = 0 then 0 else 1 in
27.                   let tag-bits = mk-Tag-Bits(raw-tag,sz-fld,sp-fld) in
28.                      if extra = 0 then
29.                         return tx-filter(tag-bits,raw-value,kind)
30.                      else
31.                         let pad-value : Spare be st pad-value = pad in
32.                            return tx-filter(tag-bits,raw-value ⌢ padding,kind)
33. )
```

Annotate

6-9. Perform any necessary concretion for the model RESPONSE and ERROR values.

10-11 Any other reply types are left unconverted.

End Annotation

```
functions
make-tag : Message-Kind → Type-Field
make-tag(kind) ≜
1.  (
2.    cases kind:
3.    ⟨NAME⟩ → [0,0,0,0],
4.    ⟨EXTERNAL-NAME⟩ → [0,0,0,1],
5.    ⟨EXTERNAL-BREAK⟩ → [0,0,1,0],
6.    ⟨USER-DATA⟩ → [0,0,1,1],
7.    ⟨RESPONSE⟩ → [0,1,0,0],
8.    ⟨ARGUMENT⟩ → [0,1,0,1],
9.    ⟨ERROR⟩ → [0,1,1,0],
10.   ⟨RESPONSE-DATA⟩ → [0,1,1,1],
11.   ⟨ERROR-DATA⟩ → [1,0,0,0]
12.   end
13. )

conc-error : Errors → Octet
conc-error(model-error) ≜
1.  (
2.    cases model-error:
3.    ⟨ARGUMENT-EXPECTED⟩ → mk-Octet([0,0,0,0,0,0,0,1]),
4.    ⟨ATTEMPTED-TO-CLOSE-UNOPENED-CHANNEL⟩ → mk-Octet([0,0,0,0,0,0,1,0]),
5.    ⟨BAD-MESSAGE-FORMAT⟩ → mk-Octet([0,0,0,0,0,0,1,1]),
6.    ⟨CANNOT-DISPATCH-MSG-TYPE⟩ → mk-Octet([0,0,0,0,0,1,0,0]),
7.    ⟨CANNOT-FREE-HANDLER⟩ → mk-Octet([0,0,0,0,0,1,0,1]),
8.    ⟨CANNOT-FREE-OLD-HANDLER⟩ → mk-Octet([0,0,0,0,0,1,1,0]),
9.    ⟨CANNOT-FREE-WORKSPACE⟩ → mk-Octet([0,0,0,0,0,1,1,1]),
10.   ⟨CANNOT-ROLLBACK⟩ → mk-Octet([0,0,0,0,1,0,0,0]),
11.   ⟨CLOSE-READ-FAILED⟩ → mk-Octet([0,0,0,0,1,0,0,1]),
12.   ⟨CLOSE-WRITE-FAILED⟩ → mk-Octet([0,0,0,0,1,0,1,0]),
13.   ⟨DID-NOT-REPLACE⟩ → mk-Octet([0,0,0,0,1,0,1,1]),
14.   ⟨DRIVER-NOT-FOUND⟩ → mk-Octet([0,0,0,0,1,1,0,0]),
15.   ⟨EXPECTED-NAME⟩ → mk-Octet([0,0,0,0,1,1,0,1]),
16.   ⟨INCOMPLETE-PATH⟩ → mk-Octet([0,0,0,0,1,1,1,0]),
17.   ⟨INSUFFICIENT-ARGS⟩ → mk-Octet([0,0,0,0,1,1,1,1]),
18.   ⟨INVALID-ENTITY⟩ → mk-Octet([0,0,0,1,0,0,0,0]),
19.   ⟨NAME-ALREADY-EXISTS⟩ → mk-Octet([0,0,0,1,0,0,0,1]),
20.   ⟨NAME-NOT-FOUND⟩ → mk-Octet([0,0,0,1,0,0,1,0]),
21.   ⟨NO-BREAK-AT-ADDRESS⟩ → mk-Octet([0,0,0,1,0,0,1,1]),
22.   ⟨NO-ENV-HANDLER-ARGS⟩ → mk-Octet([0,0,0,1,0,1,0,0]),
23.   ⟨NO-MORE-WORKSPACE⟩ → mk-Octet([0,0,0,1,0,1,0,1]),
24.   ⟨NOT-A-WORK-HANDLER⟩ → mk-Octet([0,0,0,1,0,1,1,0]),
25.   ⟨NOT-AN-ENV⟩ → mk-Octet([0,0,0,1,0,1,1,1]),
26.   ⟨OPEN-READ-FAILED⟩ → mk-Octet([0,0,0,1,1,0,0,0]),
27.   ⟨OPEN-WRITE-FAILED⟩ → mk-Octet([0,0,0,1,1,0,0,1]),
28.   ⟨UNHANDLED-INTERRUPT⟩ → mk-Octet([0,0,0,1,1,0,1,0]),
29.   end
30. )
```

```
conc-response : Responses → Octet
conc-error(model-response) ≜
1. (
2.    cases model-error:
3.    ⟨CLOSE-READ-SUCCESSFUL⟩ → mk-Octet([0,0,0,0,0,0,0,1]),
4.    ⟨CLOSE-WRITE-SUCCESSFUL⟩ → mk-Octet([0,0,0,0,0,0,1,0]),
5.    ⟨HANDLER-ADDED-OKAY⟩ → mk-Octet([0,0,0,0,0,0,1,1]),
6.    ⟨HANDLER-REMOVED-OKAY⟩ → mk-Octet([0,0,0,0,0,1,0,0]),
7.    ⟨HANDLER-REPLACED-OKAY⟩ → mk-Octet([0,0,0,0,0,1,0,1]),
8.    ⟨NAME-ADDED-OKAY⟩ → mk-Octet([0,0,0,0,0,1,1,0]),
9.    ⟨NAME-REMOVED-OKAY⟩→ mk-Octet([0,0,0,0,0,1,1,1]),
10.    ⟨OPEN-READ-SUCCESSFUL⟩ → mk-Octet([0,0,0,0,1,0,0,0]),
11.    ⟨OPEN-WRITE-SUCCESSFUL⟩ → mk-Octet([0,0,0,0,1,0,0,1]),
12.    end
13. )
```

Note in the two functions above that produce concretions for errors and responses there is *no* "others" clause in the "case" statement. This is because the model defines all known errors and responses. Model outputs produced by the monitor are totally defined. Any input to the model is different however because inputs can be incorrect.

```
functions
conc-name : token → conc-Name
conc-name(id) ≜
1. (
2.    let res : conc-Name be st id = retr-token(res) in
3.       res
4. )

conc-break-kind : Breakpoint-Kind → Octet
conc-break-kind(model-kind) ≜
1. (
2.    cases model-kind:
3.    ⟨LOCAL⟩ → mk-Octet([0,0,0,0,0,0,0,0]),
4.    ⟨GROUP⟩ → mk-Octet([0,0,0,0,0,0,0,1]),
5.    ⟨DELAYED⟩ → mk-Octet([0,0,0,0,0,0,1,0]),
6.    ⟨GLOBAL⟩ → mk-Octet([0,0,0,0,0,0,1,1])
7.    end
8. )

conc-break : Break-Identity → Octet+
conc-break(model-break) ≜
1. (
2.    let mk-Break-Identity(kind,id) = model-break in
3.       let conc-id : conc-Break-Identity be st conc-id = conc-data([id]) in
4.          let conc-bkind : conc-Break-Kind = conc-break-kind(kind) in
5.             conc-bkind ⌢ conc-id
6. )

conc-tok : token → Octet
conc-tok(tok) ≜
1. (
2.    let res : conc-token be st tok = retr-token(res) in
3.       res
4. )

conc-data : token+ → Octet+
conc-data(tokens) ≜
1. (
2.    let head = hd tokens,rest = tl tokens in
```

```
3.        if rest = [] then
4.           [conc-tok(head)]
5.        else
6.           conc-tok(head) ⌒ conc-data(rest)
7.  )
```

```
tx-filter : Tag-Bits × Octet+ × Message-Kind → [Octet+]
tx-filter(tag,octets,kind) ≜
1.  (
2.     cases kind:
3.     ⟨USER-DATA⟩,⟨EXTERNAL-BREAK⟩,⟨ERROR-DATA⟩,
4.     ⟨RESPONSE-DATA⟩,⟨ERROR⟩,⟨RESPONSE⟩ →
5.        tag ⌒ octets,
6.     others →
7.        nil
8.     end
9.  )
```

Annotate

3-5. User data,external break and error/response data is left in target native form. Error and response data has already undergone concretion by this time.

End Annotation

# Annex G

# Environmental Requirements

Given that the model described is targetted toward embedded systems development and how the monitor might be used in a real-life situation there are certain things which both PUT and monitor require of the debug agent.

It must be stressed here that the PUT is *not* a slave of the monitor server.

The monitor is designed to be minimally intrusive but to also act as a safety net when the PUT does run amok. However, when the PUT is executing "normally" there is nothing indicate that the monitor server itself exists, albeit for the possible use of user services such as entity creation, entity destruction and user input-output.

### The Debugging Environment

In Chapter 5 on pages 84-85, the steps required to initialise the PUT from the monitor's standpoint were examined. This included various conditions such as establishing a context and termination code which had to be satisfied to allow the successful execution of the PUT. Once the PUT is actually executing there has to be something that allows the PUT to initialise *itself.* The initialisation may do what has been described in the preceding paragraphs but it must obey the following (minor) restrictions if the monitor is to be allowed to do any debugging:—

(1) The monitor makes no requirement on the hardware facilities used by the PUT.

(2) If it is desired to use the monitor facilities for debugging the PUT, the PUT initialisation must satisfy the following pre-conditions:—

a. The monitor must be provided with an exclusive communications channel to the debug agent.

b. The PUT should not interfere with the break event mechanism used for a particular monitor implementation, otherwise no continuation in the model is valid from that time forward.

### Communication Requirements

For sensible *distributed* debugging to occur, the target monitors must agree (under the control of the debug agent) which communications medium is to be used by the monitor for sending and receiving group or global break event messages. If no such agreement exists then the consequences are that external breakpoint messages are *not* received by other monitors due to the fact that they are listening on (possibly) different channels. This suggests that the debug agent co-ordinates the establishment of external break events in a manner such that every target involved in a debugging session be listening to the same communications medium.

(1) A channel of communication must exist between debug agent and monitor server. Given that the messages sent by the debug agent are directed to particular target machine network addresses there must be a route that is *reachable* between the two machines.

(2) The communication channel is reliable.

### Program Under Test Requirements

The PUT is also subject to the following constraints if it is to be able to be debugged by the monitor:—

(1) The use of entity registration/deregistration facilities must abide by the concretions used to represent names in the namespace model.

(2) Entity names that may represent a thread of activity within the PUT must be unique (on the target they were created on) and their names must be valid six octet concretions.

(3) The debugger/debug client/debug agent is responsible for mapping these values to (possibly) human readable strings.

NOTE It is the responsibility of the PUT to ensure that the subordinate entities execute their finalisation and terminate *before* the PUT entity finalises and terminates.

### Debug Agent Requirements

The debug agent, much like the PUT, is subject to certain monitor requirements:—

(1) If the handler to be invoked is a model handler (e.g. an *Env-Handler*), the argument pack must be in target-independent concretion form for the purposes of communication.

(2) Argument packs to non-model handlers may be passed in any form.

(3) The debug agent requires configuration information. Such configuration descriptions are out of the scope of this thesis but they effectively amount to descriptions of the namespaces known to be present on the target machine. Any machine network addresses are also required so that communication can be initially established between the debug agent and monitor!

(4) During a switch communication channel protocol, the debug agent must listen to *both old* and *new* channels. If a monitor *Open-Write* fails, then the monitor reply will be sent using the *old* channel, otherwise the *new* channel. The method in which the monitor *Open* and *Close* handlers operate means the debug agent must be able to reconstruct the session when channels are switched by being receptive on both channels until a complete switch has taken place.

(5) The debug agent cannot construct messages larger than the MTU size. The complications that message fragmentation causes have been avoided by limiting the size of a request to that supported by the given data link layer. This reduces the complexity of the debug agent and more importantly, the monitor.

(6) No message sent to the monitor may require a response which will be greater than the MTU size.

(7) If the PUT is running on the monitor and a *Close-Session* request issued then the PUT*must* first be halted.

NOTE *Global* break messages to the monitor are needed to ensure that the PUT is halted. To ensure a "quick halt", two global break messages can be sent to the monitor to ensure that the state of execution goes from RUNNING state to a STOPPED state.

NOTE By using the existing global break mechanism, it is possible for the PUT to be *safely* shutdown *if* all finalisation code for each entity executes and normal termination occurs.

(8) User data interaction may only occur when the PUT is active.

(9) The debug agent *must not* send user data at any other time as this will result in erroneous behaviour.

(10) If a break event occurs on an entity known to have been previously removed, the debug agent must decide to:—

i. Halt the PUT and reconfigure according to the add/remove entity message received after the break event.

ii. OR Continue the PUT, updating its configuration at some later time.

# Annex H

# Debugging Methodologies

While debugging does not have a systematic, fail-safe methodology that allows for the easy location and isolation of faults as well as their correction, there have been a number of reasoning processes suggested that may help the debugger of the code focus more clearly on the problem at hand. Myers [Mye79] identifies five methodologies that may be applied to isolate and correct errors in a software system:—

(1) *Brute Force*—debugging by brute force is by far the least effective means of locating bugs. This method usually combines a number of poorly co-ordinated debugging techniques. The lack of co-ordination makes the process of debugging degrade into a "hit and miss" activity where bugs may be found by sheer luck and not through sound reasoning.

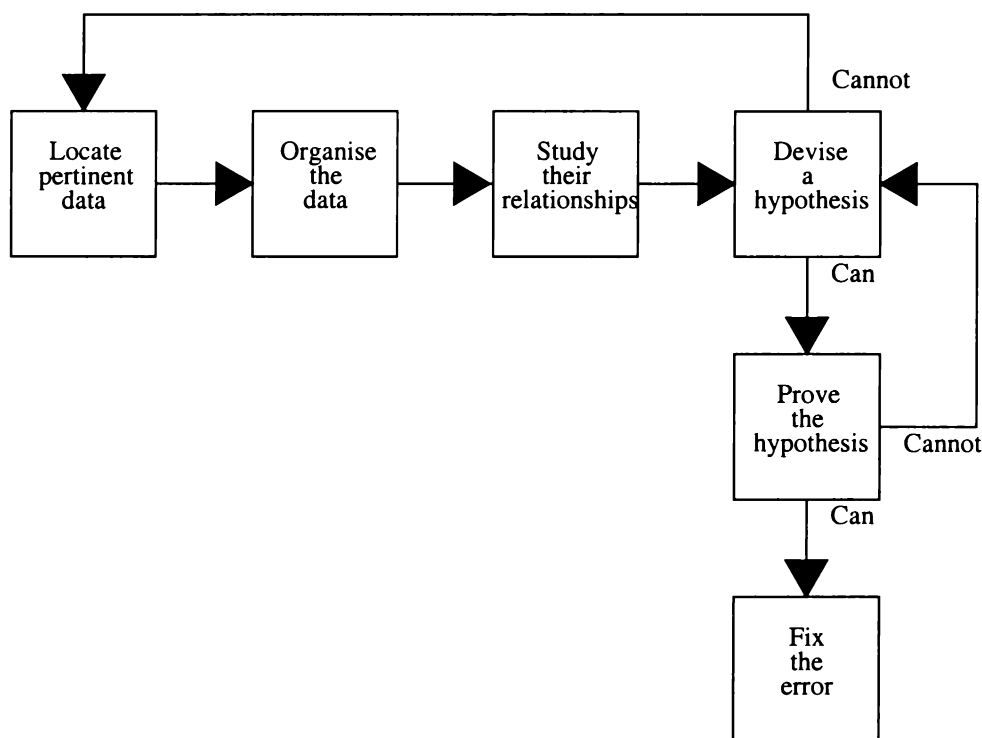



Fig. H.1 *The inductive debugging process*

(2) *Induction*—The inductive process takes particular symptoms and attempts to make inferences from these symptoms that cause the error. The symptoms may be induced by the execution of several test cases. These test cases indicate how a program performed, what was performed correctly to specification and what was performed incorrectly. This also involves test data which caused error conditions to occur. Data is then organised to factor out patterns and

to search for contradictions. Hypotheses are then generated and an attempt is made to prove them by seeing whether the hypothesis about the error is supported by the test data.

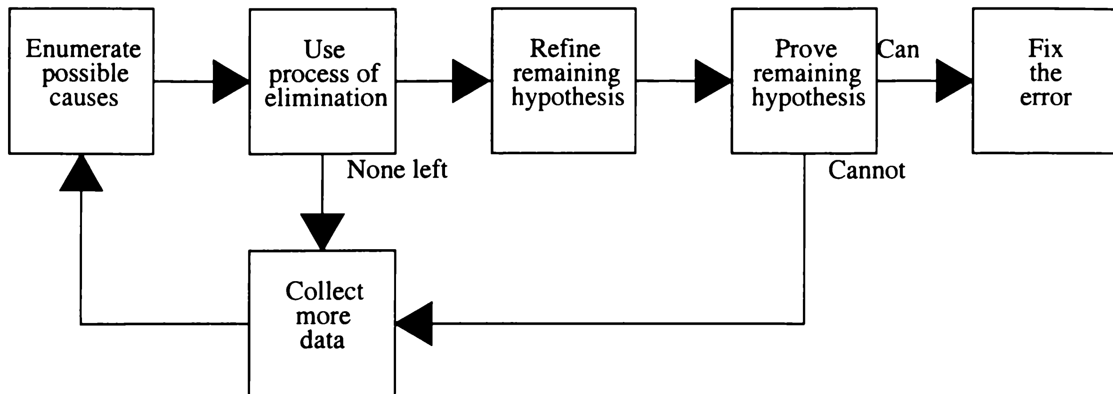



Fig. H.2 *The deductive debugging process*

(3) *Deduction*—The process of deduction proceeds from general theories or premises regarding program behaviour. Through the use of test cases, unlikely theories are eliminated until the remaining hypothesis is refined and proven consistent with the behaviour of the test case.

(4) *Backtracking*—Program backtracking reverses the execution of the program from the point where an incorrect result was produced until a state is reached where the program logic went astray.

(5) *Testing*—Program testing is used in a supporting role rather than as a stand alone method of debugging. Its focus lies in writing test cases which narrow down the cause of error. The testing methodology tends to get used during the induction/deduction methods that were explained above.

In addition to the above forms of debugging reasoning, Cheung [Che90] identifies top-down and bottom-up debugging as additional methods. These two methods rely on modular system design as the key to allowing debugging to take place. This structure can be used to narrow down the area of interest through step-wise refinement. Working software in other modules can be ignored with effort concentrated on the module whose behaviour is erroneous.

In top-down debugging the programmer considers program behaviour as a whole and successively identifies erroneous modules. The erroneous modules then become the focus of attention and detailed behaviour within each module can then be examined. This is iteratively applied until the error is located and a correction is made.

In bottom-up debugging error correction is viewed in such a way that focus is first given to each module in isolation. Each module is debugged in succession and then combined with other debugged modules. The interactions between these modules is then debugged until the system is "error free".

The process model description given by Araki [Ara91] is nearly equivalent to the process of induction that is described by Myers [Mye79]. The Araki process model breaks down debugging into five steps:—

(1) *Hypothesis set* are proven facts about the properties exhibited by the program given its specification as well as errors, their supposed cause and the potential solution to correct these errors.

(2) *Hypothesis set modification* refers to the process of generation, refinement and authentication of hypotheses. Hypotheses are generated by means of known facts about behaviour as well as from other existing hypotheses. Debugging is more than just localising and correcting—it is about knowing what you are doing and how the application should behave. Hypothesis refinement constrains or adds further detail to the hypothesis by means information gained through hypothesis verification. Verification of an hypothesis changes its truth status, hypotheses that are verified as true become facts.

(3) *Hypothesis selection* is a strategy used to choose the most likely hypothesis to attempt to verify. Selecting the best hypothesis can be done using such tactics as narrowing down where the error occurred (in a particular class, method or statement within a method). Another method is simplifying the error by attempting to find a less complex condition that causes the same erroneous behaviour.

(4) *Hypothesis verification* uses techniques to verify the hypothesis by examining the program and its behaviour. The techniques are outlined below.

Verification of hypotheses may be accomplished by the use of any of the four following techniques, static analysis, dynamic analysis, semi-dynamic analysis and program modification:—

a. *Static analysis* includes the analysis of syntactic and semantic properties of a program. Checking program structures, dependencies, module interfaces, type consistency and formal proofs of correctness may be used to gather evidence to support and generate hypotheses.

b. *Dynamic analysis* is the activity many programmers associate with "debugging". Test cases may be executed to examine behaviour which may support or refute hypotheses about the cause of error. Debugging tools are often used for the purpose of examining test case data and program behaviour.

c. *Semi-dynamic analysis* is a hybrid of the verification techniques presented above. It involves the simulation and execution by the use of computational modeling.

d. *Program modification* involves the modification of a suspect area of code to determine whether there is a relationship between the modification and the error occurrence, does it "fix" the error or does it cause no change in behaviour. This verification method is somewhat haphazard at best because understanding between why a modification that fixed the error and the cause of the original error may not be apparent.

## Debugging Techniques

Fault localisation is one of the most basic steps taken in the process of debugging program code. There are a number of techniques that are commonly used to localise faults in the code:—

a. *Output Debugging*—the most commonly used and most primitive debugging technique, output debugging is easy (if not somewhat tedious) to implement. The basic method is to place output statements at selected positions within the program. The output may consist of written messages that reveal the code's position of execution, and, in addition the values of variables at the particular position of execution.

b. *Tracing*—takes advantage of *hardware*, *operating system* or *compiler* facilities that allow code to be stepped forward one machine instruction at a time. Tracing is quite useful when analysing the flow of control from procedure to procedure whilst also examining state variables at each step to see possible causes of error at each stage.

c. *Breakpoints*—are points in the program flow where execution is suspended and whence state can then be examined or modified. Once examination or modification has been completed, program flow can be continued from the point of suspension.

d. *Assertion*—is a more powerful form of breakpoint that involves the run-time verification of a given predicate. The program is suspended if the predicate evaluates to some truth value. Thus assertion behaviour may be likened to a conditional breakpoint.

e. *Replay/Reverse Execution*—some debugging tools allow the user to capture a history of relevant execution information. When the error occurs this history information can be replayed and analysed in a controlled manner. This level of control over execution may allow the fault to be more quickly found and corrected. Also related to this concept is the idea of reverse execution. When an error occurs the debugger can, by use of history information saved, backtrack from the error to a state where it can be determined that the logic went astray and the error began to manifest itself.

f. *Monitoring*—is the capturing of useful data during execution. This data can then be used in such ways as display (via graphics or animation), replay (as described above) or further analysis by the debugger to detect errors.

### The Limitations of these Techniques

The above techniques fall under the category of *dynamic analysis.* Dynamic analysis is execution oriented, test cases are executed to determine whether the expected behaviour is exhibited. For example, the most commonly used debugging technique is the use of output statements to log program state. However, all techniques presented exhibit problems and their usage should be tempered with knowledge of these problems.

Output debugging could be considered to be used in two areas of the debugging process, hypothesis selection and hypothesis verification. For example, when a program crashes the programmer may want to narrow down the position of the occurrence of the fatal error and then when the error is localised, use further output statements to provide debugging information to verify hypotheses about the error. Output debugging, however, for all its apparent success has problems. The technique is unstructured. There is a general tendency to place output statements throughout the code just to track where the fault *is.* Subsequent output statements are required to determine why the fault is occurring by means of displaying potentially offending program states. Output debugging may also cause further errors. In the case of timing related errors, output can alter the entire behaviour of the program causing the fault to be masked. This is particularly true where concurrency is concerned.

Replay, controlled execution and reversible execution promise much in that they allow easier analysis of erroneous behaviour. The major problems with replay and reverse execution is the choice of events that are significant enough to be part of the history log. In the case of parallel debuggers the choice of events is generally restricted to inter-process communication events. Controlled execution also has problems when it comes to the analysis of timing-related errors—erroneous behaviour that may have occurred at full speed may be completely masked on a slowed down execution of the program. A further complication is the decision that needs to be made when choosing events to be captured for future replay. Some operations such as input-output are not easily reversible due to the asynchronous occurrence of such events being essentially non-deterministic.

Monitoring in terms of gathering useful information during a program's execution can cause problems by just the sheer amount of data that can be generated by a program. At the cost

of further processing this data can be filtered so that extraneous information not required can be discarded. The act of acquiring data, inevitably may also change the program's behaviour negating the usefulness of monitoring.

# Annex I

# The Rules of Operational Decomposition

## The Rules of Decomposition

The process of operational decomposition derives an executable program by a transformational approach. Once the transformation from specification to explicit form has been made, various proof obligations need to be discharged to determine whether the derivation satisfies the *pre-condition* and establishes the *post-condition.*

The executable program derived consists of *statements* that establish the state of the system through assignment, a sequence of statements, selection statements and the iteration of sequences of statements.

Specifications of the form:—

a. $P[w \backslash \overleftarrow{w}] \wedge w = E[w \backslash \overleftarrow{w}] \Rightarrow Q$

**ext wr** $w$
  **rd** $r$
**pre true**
**post** $w = E[w \backslash \overleftarrow{w}]$

This expression refines to an assignment statement:—

**ext wr** $w$
  **rd** $r$
**pre true**
  $w := E$
**post** $w = E[w \backslash \overleftarrow{w}]$

The semicolon rule is used to derive sequences of statements from specifications of the form:—

a. $P \Rightarrow P_1$

b. $P[w \backslash \overleftarrow{w}] \wedge Q_1 \Rightarrow P_2$

c. $\exists i \in State \cdot P[w \backslash \overleftarrow{w}] \wedge Q_1[w \backslash i] \wedge Q_2[\overleftarrow{w} \backslash i] \Rightarrow Q$

**ext wr** $w$
  **rd** $r$
**pre** $P$
**post** $Q$

This works by breaking specifications down to smaller sub-goals that ultimately establish a goal.

**ext wr** $w$
  **rd** $r$
**pre** $P_1$
**post** $Q_1$

```
;
ext wr w
    rd r
pre P_2
post Q_2
```

The selection rule is used to derive a conditional statements from specifications of the form:—

a. $P \wedge cond \Rightarrow P_t$

b. $P \wedge \neg cond \Rightarrow P_e$

c. $P[w \backslash \overleftarrow{w}] \wedge cond[w \backslash \overleftarrow{w}] \wedge Q_t \Rightarrow Q$

d. $P[w \backslash \overleftarrow{w}] \wedge \neg cond[w \backslash \overleftarrow{w}] \wedge Q_e \Rightarrow Q$

```
ext wr w
    rd r
pre P
post Q
```

The decomposition establishes cases where part of the post-condition can be established. Case expressions refine in a similar manner to their case statement counterparts in the executable subset of *VDM-SL*.

```
if cond then
   ext wr w
        rd r
   pre P ∧ cond
   post Q
else
   ext wr w
        rd r
   pre P ∧ ¬cond
   post Q
```

The last rule is that which applies to iteration of a statement sequence from specifications of the form:—

a. $P \Rightarrow inv$

b. $\neg guard[w \backslash \overleftarrow{w}] \wedge inv \wedge (var \vee iden) \Rightarrow Q$

```
ext wr w
    rd r
pre P
post Q
```

The *inv* is preserved by the loop body and the *guard* is defined over the body of the loop. The *var* establishes the conditions of loop termination. The use of *iden* is the identity relation. Of the four iteration constructs provided by the executable subset of *VDM-SL* the while loop is the most versatile but it requires careful consideration of the invariant and variant relations.

```
while guard do
   ext wr w
        rd r
   pre guard ∧ inv
   post inv ∧ var
```

# Annex J

# Glossary

## Technical Definitions

*Big Endian* A method by which an object greater than one octet in size is addressed in such a way that the most significant component of that quantity is at the lowest address. (See also *little endian.*)

*Break Event* A local or externally signalled event that, if conditions are satisfied, will cause a *breakpoint* that halts the program under test. (See also *Event.*)

*Breakpoint* A breakpoint halts the progress of the program under test only if the break event that occurred satisfies the criteria specified for that break event specification.

*Bug* See *fault.*

*Channel* An abstraction used to model a communication line. A channel can be opened, closed, read and written to.

*Compiler* A tool that takes well-formed source program text and translates this into some other (possibly executable) form.

*Co-routine* Components of an executing program that may transfer execution between themselves. The transfers may either occur synchronously through some explicit transfer of control mechanism or asynchronously in response to an interrupt request. A co-routine has state (local variables and closure) that persists even after a transfer takes place. Because of this a co-routine that is resumed after it transferred previously will recommence its execution at the position where the transfer originally took place.

*Data Refinement* Also known as *Data Reification.* The process of type realisation where a VDM datatype is given a real representation which is used as part of some implementation of a specification. For example, a VDM sequence type could be implemented as a linked list. The process of data refinement is closely related to *operation decomposition.*

*Datagram* A message treated by the network layer as an isolated unit.

*Debugging* The process of locating, analysing and correcting faults in a program.

*Debug Agent* A software entity executing on a host that implements the debugging back-end protocol which allows the *Debug Client* to access and possibly modify the resources of the target system. The debug agent makes requests through some communications medium to the *Monitor Server* on the target system, all targets being treated identically.

*Debug Client* A software entity that allows a user to access resources via the debug agent software. Such software provides a user interface to manipulate target resources via the debug agent.

*Debug Server* See *Monitor*

*Dialogue* The component of a system which provides state components of an interaction. It is responsible for the translation of one set of states to another sets of states. (See also *Debug Agent*)

*Distributed System* A computing architecture that is characterised by the loose coupling of its computing elements with each computing element containing its own local memory and sharing the contents of that local memory only by means of message passing.

*Dynamic Configuration* The ability to handle both changes in hardware and software during system run-time. The *monitor* uses the *namespace* model to support the dynamic configuration (addition and removal) of software and hardware components.

*Embedded System* A computer system is said to be embedded if it is a permanent component of some larger system.

*Entity* A generic term used to describe an active data object that has not only state but also execution context. Examples of entities include co-routines, tasks, threads and processes.

*Explicit Specification* see *Operational Decomposition*

*Event* Any locally or externally sourced message that has no corresponding request by the debug agent.

*Fault* A fault is an error or accidental condition that causes a program to fail to perform as specified.

*Functional Core* Those hardware and software components responsible for providing application modelling functions.

*Functional Core Adaptor* Those hardware and software components that provide and interface to the user's *system environment.*

*Global Break* A state whereby the cause of halting of the program-under-test is due to a global break event satisfying a global break event specification.

*Group Break* A state whereby the cause of halting of the program-under-test is due to a group break event satisfying a group break event specification.

*Global State* In a distributed system, global state is the union of all state local to individual processors and any messages in transit between these processors.

*Implicit Specification* A form of specification in which the functionality is specified in terms of contract between caller and callee of functions and operations.

*Little Endian* A method by which an object greater than one octet in size is addressed in such a way that the least significant component of that quantity is at the lowest address. (See also *big endian.*)

*Local Break* A state whereby the cause of halting of the program-under-test is due to a local break event satisfying a local break event specification.

*Mission Critical System* The computer software and hardware components of a system whose function is critical in and defined by the environment in which it functions.

*Monitor Server* See also *Monitor*

*Monitor* A software entity that executes on a target system as an *actor*. When requests are received from the debug agent, the monitor services these and sends a reply back to the debug agent (and usually, therefore, subsequently to the debug client).

*Namespace* The namespace is an object that contains state and a mapping from names to actions or names to other namespaces. It is used to model the *functional core* of the monitor server.

*Octet* A sequence of binary digits whose length is eight bits. It is the fundamental unit of digital communication.

*Operating System* An operating system consists of those software and firmware components running on some hardware which are required in order to provide common functionality of application software under different hardware configurations.

*Operation Decomposition* The process of taking an implicit specification and deriving this into a form that is executable by some interpreter or possibly useful to post-processing by a code generator.

*Probe Effect* The Probe Effect is the resulting disturbance of operation of some system caused by the measurement operations performed.

*Program-Under-Test (PUT)* Any software running on the target system that is not directly under monitor control.

*Protocol* A set of rules formulated to control the exchange of data between two communicating parties.

*Real Time System* A system whose correctness of operation is dependent on inputs and the time at which those inputs occur.

*Remote Procedure Call* Invocation of some function or operation on a target machine different from the one invoking the function or operation.

*Synchronous Transmission* A method of transmission whereby the message being sent is framed by synchronisation markers which the receiver uses to retrieve the message.

*Target* A computer system upon which the monitor server and program under test execute.

*Target-dependent* This refers to values or functionality the semantics of which are determined by the target. In practice any target-dependent function or operation will encompass a very few machine instructions.

*VDM-SL* Vienna Development Method Specification Language.

*Wire Protocol* A group of data structures and operations that are used to encode and decode model messages to a representation suitable for communication along a communications medium. For example, Ethernet, serial line, etc.

# Annex K

# References and Bibliography

**References**


[Ara91] Araki K, Furukawa Z & Cheng J, *A General Framework for Debugging*, IEEE Software, 5:14-20, May 1991.

[Ben88] Bentley JL, *More Programming Pearls : Confessions of a Coder*, Addison-Wesley Pub. Co., Reading, Mass., 1988.

[Bur94] Burgess P, *A Testbed for Embedded Systems*, Ph.D. Thesis University of St. Andrews, St. Andrews, Fife, KY16 9SS, United Kingdom, 1994.

[Car83] Cargill TA, *The Blit Debugger*, The Journal of Systems and Software, 3:277-284, March 1983.

[Car84] Cargill TA, *Debugging C Programs With the Blit*, AT&T Bell Laboratories Technical Journal, 63(8):1633-1647, October 1984.

[Car85] Cargill TA, *Implementation of the Blit Debugger*, Software—Practice and Experience, 15(2):153-168, February 1985.

[Car86] Cargill TA, *Pi: A Case Study in Object-Oriented Programming*, Proceedings of OOPSLA '86, ACM Press, 21(11):350-360, Portland, Oregon, September 29–October 2 1986.

[Cha85] Chandy KM & Lamport L, *Distributed Snapshots: Determining Global States of Distributed Systems*, ACM Trans. on Comp. Sys., 3(1):63-75, February 1985.

[Che90] Cheung WH, Black J & Manning E, *A Framework for Distributed Debugging*, IEEE Software, 1:106-115, January 1990.

[Coh81] Cohen D, *On Holy Wars and a Plea for Peace*, IEEE Computer, 14(10):48-54, October 1981.

[Com93] Comer DE, Stevens DL, *Internetworking with TCP/IP, Volume III Client-Server Programming and Applications*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1993.

[Den92] Deneau T, *Universal Debugger Interface Specification, Version 1.2*, Technical Report, Advanced Micro Devices (AMD), 1st May, 1992.

[Gram96] Gram C & Cockton G, *Design Principles for Interactive Software*, Chapman & Hall (London), pp 93 et seq., 1996.

[Hab90] Haban D & Wybranietz D, *A Hybrid Monitor for Behaviour and Performance Analysis of Distributed Systems*, IEEE Trans. on Soft. Eng., 16(2):197-211, February 1990.

[Hal88] Halsall F, *Data Communications, Networks and OSI, 2nd ed.*, Addison-Wesley Pub. Co., England, 1988.

[IFAD99] Peter Gorm Larsen et. al., *The IFAD VDM-SL Language & VDM-SL Toolbox User Manual*, Available via download at http://www.ifad.dk, 1999.

[IF++99] Peter Gorm Larsen et. al., *The IFAD VDM++ Language & VDM++ Toolbox User Manual*, Available via download at http://www.ifad.dk, 1999.

[ISO94] Andrews D et. al., *Information technology—Modula-2*, Technical Report IS 10514:1994, ISO, 1994.

[Joy87] Joyce J, Lomow G, Slind K & Unger B, *Monitoring Distributed Systems*, ACM Trans. on Comp. Sys., 5(2):121-150, May 1987.

[Kir83] Kirrmann H, *Data Format and Bus Compatibility in Multiprocessors*, IEEE Micro, (8):32-47, August 1983.

[Kün86] Kündig AT, *A Note on the Meaning of "Embedded Systems"*, Lecture Notes in Computer Science : Embedded Systems, Springer-Verlag, 284:1-5, 1986.

[Lam78] Lamport L, *Time, Clocks and the Ordering of Events in a Distributed System*, Comm. ACM, 21(7):558-565, 1978.

[Lar98] Peter Gorm Larsen & John Fitzgerald, *Modelling Systems: Practical Tools and Techniques in Software Development*, Cambridge University Press, 1998.

[Mat93] Mattern F, *Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation*, Journal of Parallel and Distributed Computing, 18:423-434, 1993.

[McD89] McDowell C & Helmbold D, *Debugging Concurrent Programs*, ACM Computing Surveys, 21(4):593-623, December 1989.

[Mil76] Milne R & Strachey C, *A theory of programming language semantics*, Chapman & Hall (London), 1976.

[Mye79] Myers GJ, *The Art of Software Testing*, Wiley, 1979.

[Org73] Organick EI, *Computer System Organisation: The B5700/B6700 Series*, Academic Press, Inc., 1973.

[Pan93] Pancake CM & Netzer RHB, *A Bibliography of Parallel Debuggers, 1993 Edition*, Proceedings of the Workshop on Parallel and Distributed Debugging, ACM Press, 28(12):169-186, San Diego, California, 17-18 May 1993.

[Ram92] Ramsey N & Hanson DR, *A Retargetable Debugger*, Proceedings of the ACM Sigplan '92 Conference on Program Language Design and Implementation, ACM Press, 27(7):22-31, San Francisco, California, 1992.

[Ray92] Raynal M, *About logical clocks for distributed systems*, Operating Systems Review, 26(1):41-48, January 1992.

[Red89] Redell DD, *Experience with Topaz Teledebugging*, Proceedings of the Workshop on Parallel and Distributed Debugging, ACM Sigplan/Sigops, 24(1):35-44, Madison, Wisconsin, January 1989.

[Sor94] Sorel PE, Fernandez MG & Ghosh S, *A Dynamic Debugger For Asynchronous Distributed Algorithms*, IEEE Software, 1:69-76, January 1994.

[Sta88] Stankovic JA, *Misconceptions About Real-Time Computing—A Serious Problem for Next-Generation Systems*, IEEE Computer, 10:10-19, October 1988.

[Ste90] Stevens WR, *UNIX Network Programming*, Prentice-Hall, 1990.

[Tsa90a] Tsai Jeffrey JP, Kwang-Ya Fang & Horng-Yuan Chen, *A Noninvasive Architecture to Monitor Real-Time Distributed Systems*, IEEE Computer, 3:11-23, March 1990.

[Tsa90b] Tsai Jeffrey JP, Kwang-Ya Fang, Horng-Yuan Chen & Yao-Dong Bi, *A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging*, IEEE Trans. on Soft. Eng., 16(8):897-916, August 1990.

**Bibliography**


[Ada86] Adams E & Muchnick SS, *Dbxtool: A window-based symbolic debugger for Sun workstations*, Software–Practice and Experience, 16(7):653-669, July 1986.

[And91] Andrews D & Ince I, *Practical Formal Methods with VDM*, McGraw-Hill (UK), 1991.

[Bem86] Bemmerl T, *Realtime High Level Debugging in Host/Target Environments*, Microprocessing and Microprogramming, 18:387-400, 1986.

[Bic94] Bicarregui JC, Fitzgerald JS, Lindsay PA, Moore R & Ritchie B, *Proof in VDM: A Practitioner's Guide*, Springer Verlag London Limited, 1994.

[Bru91] Bruegge B, *A Portable Platform for Distributed Event Environments*, Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, ACM Press, 26(12):184-193, Santa Cruz, California, 20-21 May 1991.

[Cla87] Clarke EM & Grümberg O, *Research on Automatic Verification of Finite State Concurrent Systems*, Ann. Rev. Comput. Sci., 2:269-290, 1987.

[Coom93] Coombes A & McDermid J, *Specifying temporal requirements for distributed real-time systems in Z*, Software Engineering Journal, 9:273-283, September 1993.

[Coop91] Cooper R & Marzullo K, *Consistent Detection of Global Predicates*, Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, ACM Press, 26(12):167-174, Santa Cruz, California, 20-21 May 1991.

[Daw91] Dawes J, *The VDM-SL Reference Guide*, Pitman (UK), 1991.

[Fid89] Fidge CJ, *Partial Orders for Parallel Debugging*, Proceedings of the Workshop on Parallel and Distributed Debugging, ACM Press, 24(1):183-194, Madison, Wisconsin, January 1989.

[Gol91] Goldsack SJ & Finkelstein ACW, *Requirements engineering for real-time systems*, Software Engineering Journal, 5:101-115, May 1991.

[Halp87] Halpern JY, *Using Reasoning about Knowledge to Analyze Distributed Systems*, Ann. Rev. Comput. Sci., 2:37-68, 1987.

[Hala92] Halang WA, *Real-Time Systems: Another Perspective*, J. Systems Software, 18:101-108, 1992.

[Jon86] Jones CB, *Systematic Software Development Using VDM*, Prentice-Hall International (UK) Ltd., 1986.

[Kav92] Kavi KM & Yang Seung-Min, *Real-time Systems Design Methodologies: An Introduction and a Survey*, J. Systems Software, 18:85-99, 1992.

[Kri89] Krishnamurthy EV, *Parallel Processing : Principles and Practice*, Addison-Wesley Singapore, 1989.

[Laz86] Lazzerini B & Prete CA, *DisDeb: An Interactive High-Level Debugging System for a Multi-Microprocessor System*, Microprocessing and Microprogramming, 18:401-408, 1986.

[Lyt90] Lyttle D & Ford R, *A Symbolic Debugger for Real-time Embedded Ada Software*, Software–Practice and Experience, 20(5):499-514, May 1990.

[Ost92] Ostroff JS, *Formal Methods for the Specification and Design of Real-Time Safety Critical Systems*, J. Systems Software, 18:33-60, 1992.

[Pik84] Pike R, Locanthi B & Reiser J, *Hardware/Software Trade-offs for Bitmap Graphics on the Blit*, Software–Practice and Experience, 15(2):131-151, February 1984.

[Rud87] Rudin H, *Network Protocols and Tools to Help Produce Them*, Ann. Rev. Comput. Sci., 2:291-316, 1987.

[Spe91] Spezialetti M, *An Approach to Reducing Delays in Recognizing Distributed Event Occurrences*, Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, ACM Press, 26(12):155-166, Santa Cruz, California, 20-21 May 1991.

[Yan94] Yang Z & Marsland TA, *Global States and Time in Distributed Systems*, IEEE Computer Society Press (Los Alamitos, CA 90720-1264, U.S.A.), 1994.

[Won92] Wong Derek, *Protocols for Debugging Distributed Embedded Systems*, New Zealand Computer Science Research Students' Conference 1992, pp 287-294, Hamilton, New Zealand, 1992.

[Won95] Wong Derek, *Adaptive Debugging for Embedded Systems*, New Zealand Computer Science Research Students' Conference 1995, pp 283-292, Hamilton, New Zealand, 1995.