1

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

National Library
of CanadaBibliothèque nationale
du CanadaCanadian Theses
on Microfiche ServiceServices des thèses canadiennes
sur microfichesOttawa, Canada
K1A 0N4

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

• Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur

STYLIANOS SARTZETAKIS

Date of Birth — Date de naissance

31-10-59

Country of Birth — Lieu de naissance

GREECE

Permanent Address — Résidence fixe

2-Stadiou str. Polichni Thessaloniki 565 33
GREECE

Title of Thesis — Titre de la thèse

TOWARDS AN INTEGRATED,
FEEDBACK-DIRECTED
SOFTWARE DEVELOPMENT ENVIRONMENT

University — Université

CARLETON UNIVERSITY, OTTAWA

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée

MASTER OF ENGINEERING

Year this degree conferred — Année d'obtention de ce grade

1985

Name of Supervisor — Nom du directeur de thèse

R.J.A. BUHR

Permission is hereby granted to the NATIONAL LIBRARY OF
CANADA to microfilm this thesis and to lend or sell copies of
the film.The author reserves other publication rights, and neither the
thesis nor extensive extracts from it may be printed or other-
wise reproduced without the author's written permission.L'autorisation est, par la présente, accordée à la BIBLIOTHÈ-
QUE NATIONALE DU CANADA de microfilmer cette thèse et de
prêter ou de vendre des exemplaires du film.L'auteur se réserve les autres droits de publication; ni la thèse
ni de longs extraits de celle-ci ne doivent être imprimés ou
autrement reproduits sans l'autorisation écrite de l'auteur.

Date

Fri, Aug 23, 1985

Signature

some SDEs, as part of the system's development support at the high level

Even though high level design testing can detect serious design problems, circular deadlocks for example, and predict system performance, it is not enough to assure software quality and effectiveness. Automated *measurement* tools are necessary to collect, easily and cost effectively, data guiding the testing process, bringing quantitative perspective in software quality measurement. Such a measurement system defines the characteristics of the information being processed, identifying critical factors that will influence the design during the *requirements phase*. The same system helps in performance evaluation during the *design phase* and feeds actual performance data from the implemented design, exercising the implementation with test runs. The same data are used to facilitate the designing process when the system runs with real input. The *verification* process does not only verify the module function and connection correctness, but the performance of the whole system as well. After the system is verified, test data can be recorded for use during the *maintenance phase* for the rest of the systems life

1.2 The Experimental Environment and its Tools

Given the high, constantly increasing complexity of today's software and the limitations of the software developers as human beings, the need for powerful and sophisticated SDEs is emerging, to relieve much of the developers' burden. Such an experimental environment is presented here, to show how the principles and ideas discussed in the previous section can be applied, and to what extent they may be applied. It was not designed by the author, but is the result of continuous efforts over the last five years. Its main part is the CAEDE system [7], which in its current state provides an environment with a unique structured design methodology (an extension to SDWA 4), an iconic design entry system, a design data base, design structure reasoning, system's performance prediction, and automatic skeleton code generation.

The undersigned hereby recommends to the Faculty of Graduate
Studies and Research acceptance of the thesis,

**TOWARDS AN INTEGRATED, FEEDBACK-DIRECTED SOFTWARE
DEVELOPMENT ENVIRONMENT**

submitted by STYLIANOS SARTZETAKIS, B.Sc., in partial fulfilment of the
requirements for the degree of

Master of Engineering

Thesis Supervisor

Chairman,
Department of Systems
and Computer Engineering

Department of Systems & Computer Engineering
Faculty of Engineering
Carleton University
August 23, 1985

Abstract

One of the pervasive concepts in Software Engineering is that of the development environment. This thesis presents the state of the art in software engineering environments, proposes a new rapid prototyping approach in the development of software, and reports on the development of a performance monitoring tool as part of an experimental software prototyping environment. The latter supports the development of complicated concurrent real-time systems ensuring that they meet their time, reliability, and complex environmental interaction constraints.

According to the proposed method, a system is designed, automatically implemented, error checked, and executed under the supervision of a special performance monitoring tool, which analyzes the implementation execution to show conformance with or divergence from the requirements specification. This sequence is repeated until the under development system meets its requirements.

Now it may be said of any kind of action that the action itself, as such, is neither good nor bad. Take, for example, what we are doing now. Neither drinking nor singing nor talking has any virtue in itself, for the outcome of each action depends upon how it is performed. If it is done rightly and finely, the action will be good; if it is done basely, bad.

PLATO, Symposium

Acknowledgments

The author wishes to express his gratitude to his supervisor Prof. Ray Buhr, for his guidance, encouragement, and support throughout the research work described in this thesis and moreover during the course work that was preceded. He is indebted to Kim Rowe, first for prodding his interest in monitoring systems performance, and following that, for the many stimulating discussions and criticisms. Deepest appreciation to his friend Maria. She helped in so many ways during the last two years, that typing and proofreading of this thesis seems only a minor help. The work done could never take place without the invaluable support from his parents. To them this thesis is dedicated.

Contents

Abstract	i
Acknowledgments	iii
Contents	iv
List of figures	vi
List of tables	vii
1 Introduction	1
1.1 Background	1
1.1.1 What is a Software Development Environment	2
1.1.2 The Need for Feedback during Development	3
1.2 The Experimental Environment and its Tools	4
1.2.1 The Dynamic Performance Monitoring system	5
1.2.2 The Target Environment	6
1.3 Objectives and Accomplishments	6
1.4 Thesis Outline	7
2 Background Motivation and Review of State of the Art	9
2.1 Software Engineering Environments	9
2.1.1 Environments Classification and Terminology Definitions	10
2.1.2 Programming Support Environments	17
2.1.3 Integrated PSEs	22
2.1.4 Design Support Environments	29
2.1.5 Software Development Environments	32
2.2 Rapid Prototyping	35
2.3 Performance Measurement and Evaluation	36
2.3.1 Performance Modelling	37
2.3.2 Performance Monitoring	38
2.3.3 Real-time Embedded Systems	42
3 A Proposed Approach	45
3.1 Origin of idea	45
3.2 An Idealized Software Development Environment	46

3.3	The Prototype Software Development Environment	54
3.3.1	The Execution Environment	55
4	Design of an Experimental Prototype	60
4.1	The Dynamic Performance Monitoring system	60
4.1.1	DPM Version 1.0	63
4.1.2	DPM Version 1.1	65
4.1.3	DPM Version 2.0	66
4.1.4	DPM Version 2.1	67
4.2	Testing	68
5	Conclusions and Recommendations	74
5.1	Conclusions	74
5.2	Future Work	75
	Bibliography	77
	Appendices :	
A	Software Engineering Environments Index	A-1
B	Example Program Listings	B-1
C	Abbreviations and Acronyms	C-1

List of Figures

2 1	Performance Tools classification	38
2 2	Performance monitoring system functions	39
3 1	The waterfall model of the software life cycle	47
3 2	The proposed use of the monitoring tool in the rapid prototyping process	47
3 3	The rapid prototyping environment with the proposed feedback connection	49
3 4	The idealized, CAEDE based SDE	50
3 5	The embedded system software	56
3 6	The mapping of the multiprocessor linear memory address space onto the various processors	59
4 1	The logic analyzer and its position in the environment	61
4 2	The ARTT laboratory hardware	62
4 3	Timing diagram for the Harmony/C example	69
4 4	The producer - consumer system design.	70
4 5	The graphics output of the DPM tool.	72
4 6	A snapshot of the SUN workstation screen during development	73
B 1	The Harmony/C program	B-2
B 2	The Ada program	B-6

List of Tables

2 1	Representative SEEs, the areas they support, and the main emphasis points	11
3 1	Comparison of the CAEDE's performance modelling tool with the monitoring tool.	53

Chapter 1

Introduction

1.1 Background

Software engineering has evolved into a very complex, difficult, and time consuming task. Software production of today's systems is one of the most sophisticated intellectual tasks. Technical production, project management and application support are parts of the software development process, of which programming is only one phase. The *software crisis* came as a result of the limited support that systems were providing during this complicated development process. Elementary programming support environments start appearing, and a continuous evolution followed. These new environments directly addressed the problem through the reduction of both time and complexity required to develop software products. Today, a large number of such environments have been built, enhanced and augmented to support every development phase.

Real-time embedded systems have further complexities associated with time constraints, complex environmental interaction, power, reliability, and space constraints. Few if any of these software engineering environments deal with the additional complexities introduced under these circumstances. This thesis deals with an environment which supports the software engineering of real-time embedded systems.

1.1.1 What is a Software Development Environment

Development begins as soon as the need for the product is identified; it ends when the implemented product is tested and delivered for operation. Software Development Environments (SDEs) support this development process, bringing system analysis, design, implementation, verification and maintenance closer together conceptually, and provide means for moving from phase to phase. Automation for the transformations is offered in an interactive, fast and effective manner. SDEs are automated environments that support the application of new methodologies to the software production. They correspond to the need for high speed, performance and reliability in today's systems, supporting the successful development of complicated software, like that of multitasking or multiprogramming systems.

New practices are introduced, since in the past, methodologies have been little help in dealing with software development and maintenance complexities. SDEs brought the transformation in the way software is manufactured, and the formalization of software development into an engineering practice. The conventional approach to software development (waterfall life-cycle model), has been challenged by the new operational (separation of problem-oriented from implementation-oriented concerns) [89], and the rapid prototyping approach. Following the conventional approach, a full blown implementation is undertaken after a long requirements specification phase; however, the time delay, and poorly understood and changing requirements result in later respecification and redesign. Typically, projects end up being out of schedule and all too often with performance problems due to poor and changing specifications.

In the operational approach, problem understanding leads to operational specifications executable by a suitable interpreter and the transformation of these specifications that follows, leads to the final system. In rapid prototyping, continuous revision on a lower level, the design level, produces the desired system.

A special kind of rapid prototyping is being proposed in this thesis. According to this method, a system's design is automatically implemented and executed under the supervision of a special performance monitoring tool, which analyzes the implementation execution to show conformance with, or divergence from, the requirements definition. The experimental environment, which supports this kind of rapid prototyping, provides means to check the high level design correctness, predict its performance, and in addition debug the implementations' low level code. Since all these fundamental functions are acting in a bottom-up fashion, providing feedback about the quality and requirements specification conformance of the system being produced, the environment has been termed *feedback directed*.

Due to the limitations existing at the time this research was taking place, systems designs were manually implemented.

1.1.2 The Need for Feedback during Development

In SDEs, for the first time, performance considerations as well as functional considerations are integral part of the design process. Up to now software performance was not considered until after its implementation or even not until acceptance testing. The main research effort was put in assuring the functional correctness of the system and in simplifying the design process. Measurement in general, was carried out in an ad hoc manner. Most of today's programming environments offer support for the validation and verification phases but do not integrate the tools for performance measurement. Moreover these phases are following the implementation phase and if it is proven that a system does not meet its minimal performance requirements, large parts of the project had to be revised or redeveloped from the scratch. SDEs attempt to integrate all development phases by automating the process of transformation between them. Tools capable of reasoning during the design phase, checking the consistency of the structure, and tools that are using modelling methods to estimate and predict systems performance, are included in

some SDEs, as part of the system's development support at the high level

Even though high level design testing can detect serious design problems, circular deadlocks for example, and predict system performance, it is not enough to assure software quality and effectiveness. Automated *measurement* tools are necessary to collect, easily and cost effectively, data guiding the testing process, bringing quantitative perspective in software quality measurement. Such a measurement system defines the characteristics of the information being processed, identifying critical factors that will influence the design during the *requirements phase*. The same system helps in performance evaluation during the *design phase* and feeds actual performance data from the implemented design, exercising the implementation with test runs. The same data are used to facilitate the designing process when the system runs with real input. The *verification* process does not only verify the module function and connection correctness, but the performance of the whole system as well. After the system is verified, test data can be recorded for use during the *maintenance phase* for the rest of the systems life

1.2 The Experimental Environment and its Tools

Given the high, constantly increasing complexity of today's software and the limitations of the software developers as human beings, the need for powerful and sophisticated SDEs is emerging, to relieve much of the developers' burden. Such an experimental environment is presented here, to show how the principles and ideas discussed in the previous section can be applied, and to what extent they may be applied. It was not designed by the author, but is the result of continuous efforts over the last five years. Its main part is the CAEDE system [7], which in its current state provides an environment with a unique structured design methodology (an extension to SDWA 4), an iconic design entry system, a design data base, design structure reasoning, system's performance prediction, and automatic skeleton code generation.

An enhancement has been made to the system, with the addition of an execution support sub-environment, and new features have been added providing dynamic performance monitoring and symbolic debugging. However, these new features are not absorbed into one integrated system.

The theme of the experimental part of this thesis is the design, development, and integration in a degree with the CAEDE (graphics part), of a prototype Dynamic Performance Monitoring system (DPM)

Monitoring tools act in an interactive manner to provide dynamic analysis of the system under design, which could be combined with static structural, functional and performance modelling of the design, in order to provide the maximum assistance during the development process. The result is an augmentation of the designer's reasoning abilities. An SDE with these features does not follow the conventional, waterfall life-cycle model anymore, being an *interactive prototyping* environment, where the user cooperates with the system to refine the design and even the specifications.

1.2.1 The Dynamic Performance Monitoring system

The developed dynamic performance monitoring and evaluation system provides an integrating function in the framework of the development environment. It is considered to be an *on-line, event driven, hardware monitor*. It combines hardware instrumentation for the collection of the appropriate data, using information about the application from the design's logical structure and the compiler, with software packages to reduce the data. The results are presented on the designer's terminal graphically during test execution. The measured parameters are: 1) processor time consumption by the various tasks, 2) rendezvous delay times, and 3) VME bus access. The observed system is not altered in any way, no software parts are replaced by test stubs (as in unit testing methods), and no software instrumentation exists in a way that alters the target's processes time or space characteristics.

1.2.2 The Target Environment

The fact that the proposed environment is intended to support real-time embedded systems, implies the existence of a target system. This target, is a multiprocessor consisting of several MC68010 boards on a VME bus. A new experimental multitasking, multiprocessing real-time operating system, the Harmony O.S., was used for application development. This execution environment was used for testing and demonstration purposes during the development of the various versions of the prototype DPM tool.

1.3 Objectives and Accomplishments

The goal of this research is to overcome some of the common problems associated with the development of real-time software. The new rapid prototyping approach redefines the role of the software engineer-developer, by providing a development environment that will free him from low level considerations, and will let him concentrate on the design and optimization of the under development system. This can only be achieved if the environment is as much automated as possible, and interactive at the same time.

Consequently, the understanding of how performance considerations can be taken care of during the development of real-time embedded systems was a major concern. The primary purpose of this project was to examine the feasibility of closing the loop from the execution to the design stage during software development, and show the implications from monitoring system design. The closing of this loop has been attempted here, by means of a real-time monitoring tool. The existence of a rapid prototyping environment in the way it is described above, is assumed, including the existence of an execution environment. A tool has been built, the DPM, which, to a certain extent, gives the designer the ability to see the implications on performance of alternative design considerations, and to evaluate his designs with the higher possible degree of accuracy. Its main advantages are 1) its accuracy, since it *measures* performance, 2) the fact that it

is not limited to measure only software characteristics but hardware as well, illustrating the degree of their integration, and 3) the fact that the designer is not required to be familiar with low level measurement methodologies (triggering on interrupts, on special addresses, sampling, averaging or monitoring at assembly level). A real-time summary of system execution indices comes to his graphics terminal automatically. DPM abstracts on behalf of the designer low level information, doing the required performance analysis

The prototype environment is in no way a 'near-product'. It is only a research vehicle to explore several issues, and should be viewed as such. The currently existing version of the DPM tool is only an intermediate one. The improvements suggested in the last chapter have to be considered, and more work has to be done implementing those ideas, in order to complete the system to a level where it will be capable of real contribution in the effort to ease the laboured software development process.

However, the environment in its current state automates many of the required development phases, facilitating the design and the performance evaluation of concurrent systems. The performance monitoring tool provides feedback from the execution environment and opens the way for more integration in the environment. The presented environment is one of the very few of its kind and unique in approach and methodology.

1.4 Thesis Outline

The next chapter is devoted entirely in an extended overview of the state of the art in the field, together with an analysis of the main issues, and a quick definition of the most frequently used terms. This was considered to be expedient in order to put performance monitoring, the main topic of the experimental part of the thesis, in context.

The first section of this chapter is a survey of the major offered software engineering environments, focusing in recent developments, and emphasizing trends and challenges. Issues discussed in this section are the need for software engineering environments,

their components and features, the realized technical concepts, their application areas and origin of ideas. The area is divided into four categories, based mainly on the part of the software life-cycle that is supported, in such a way that any of the existing and proposed environments belong to one of them. A perspective of their evolution is offered as well.

An environments index, that comes out from this review section, is contained in the first appendix. It provides a short description of the systems, notes their statuses, and associates them with their developers, the article's authors and the pages where they are described. This index frees the author from the obligation for frequent citations. The reader can easily consult this index for whatever environment is referenced but not cited. Environments are cited only at the point where they are described.

A review of the existing methodologies used to measure and evaluate system performance is given in the next section. Many of the terms related to performance modelling and monitoring, are defined in this section. A rough assessment of the basic methods is given as well.

The third chapter analyzes the proposed approach, its origins and objectives, its framework, and its strategy.

The design of the experimental prototype DPM that has been implemented for the purposes of this thesis, and as part of a research project, is detailed in the next chapter. Reports on the successive versions of the prototype implementation are also given in this chapter.

There are other ideas for further development, which could not be implemented in the limited time allowed for this project. They constitute a whole set of improvements that could bring the prototype DPM system into a more advanced stage, with regards to its role in the rapid prototyping environment. The fifth chapter is devoted to the discussion of those ideas, and reports on work to be done in the future.

Chapter 2

Background Motivation and Review of State of the Art

The software development is put in perspective in this chapter, where the history of the software engineering environments evolution is presented, and special attention is given to the new approaches. One of the prime purposes of this review was to take stock of the state of the art in software development systems, and show future trends. Recent papers on the major software engineering systems are reviewed and their main aspects are being classified and described in the next section. A review of the role of rapid prototyping in the development of software is given next, and then, an analysis of the main performance measurement and evaluation methods follows. It is also in this chapter, where a large number of the most frequently used terms is defined.

2.1 Software Engineering Environments

A software engineering environment (SEE) can be viewed as a laboratory for software development where many people are working together using tools and instruments in a fully organized manner. More than thirty such SEEs are presented in this section, referenced in more than seventy articles (all published after 1975). Nevertheless this survey is by no means exhaustive. The set of systems used, is being considered representative

of the state of the art, and as such provided the basis for the analysis. An historical retrospection is offered as well, since the classification represents inevitable the advance of the software technology over the time (the last twenty years).

Direct comparison of functionality proved to be difficult, if not impossible, due to the different goals of the environments' designers. For this reason, the environments are classified into four categories. This aims in the identification and description of the basic features, and in the explanation of the way in which the more advanced environments in the category act, in comparison with the more elementary ones.

Table 2.1 illustrates the extend of support that some characteristic SEEs offer over the software life cycle (waterfall model).

2.1.1 Environments Classification and Terminology Definitions

Depending on the part of the waterfall life-cycle model each one of them supports, the year of appearance, and the goals of its designers, SEEs are classified here, into the following categories

- Programming Support Environments (PSEs)

PSEs start appearing at about 1970, with the introduction of software engineering.

There were nothing but a collection of a few tools at the beginning, but they grew up to become unmanageable huge collections of heterogeneous pieces of software.

We examine some of them in two separate subsections:

- Elementary PSEs , and
- Elaborate PSEs

- Integrated PSEs (IPSEs)

- Design Support Environments (DSEs)

Environments	Software Development Phases						
	RQ	DS	GR	PR	TV	PM	MT
AIDES	-	o	•	o	o	•	-
APSE	-	o	-	•	o	-	-
Arcturus	o	o	-	•	o	-	•
ARGUS	-	o	o	•	•	o	o
CADAS	•	o	-	•	•	-	-
CADES	•	o	-	o	o	•	-
CAEDE	o	•	•	o	-	-	-
Cedar	-	-	•	•	•	-	-
COSY	-	o	-	o	o	-	-
CNET	-	o	-	o	o	-	-
DICE	-	o	-	•	o	-	-
DREAM	-	•	-	o	o	-	-
DSEE	-	-	-	•	•	•	o
Gandalf	-	o	o	•	-	o	-
INTERLISP	-	o	-	•	•	-	-
Magpie	-	-	o	•	•	-	-
MELBA	-	•	•	•	•	-	-
MML	-	-	-	o	o	-	-
PECAN	-	-	o	•	•	-	-
PegaSys	-	•	•	•	-	-	-
PDS	-	-	-	o	o	-	-
Prometheus	o	•	o	o	o	-	o
PWB/UNIX	-	-	-	o	o	o	•
SIMPOS	-	o	-	•	-	o	o
Smalltalk	-	o	•	•	o	-	-
SPP	-	-	-	o	o	o	-
SWB	o	o	-	o	o	•	o
UNIX™	-	-	-	•	-	o	o

- not supported, o : partly supported, • : main emphasis.

RQ : Requirements specification, DS : Design, GR : Graphics support of any Kind,
 PR : Programming, TV : Testing and Verification,
 PM : Project Management, MT : Maintenance.

Table 2.1: Representative SEEs, the areas they support, and the main emphasis points

IPSEs and SDEs came as an answer to the problem of managing complex software constructs and as a result of the advance in the area of interface technology. What they have in common is the lack of integration and overall consistency.

- Software Development Environments (SDEs)

SDEs are characterized by the broad area of the software life-cycle they support, especially in the early and late phases, namely requirements – design, and verification – maintenance. They are characterised by a high degree of integration, granularity, and reusability in such a way that the waterfall model starts to be in dispute, and we cannot tell anymore what phase of the model is supported.

The difference *between* the identified categories is in how wider is the support of the software product over its life-cycle, whereas *inside* the categories, environments differ in how stronger is the support they offer.

The continuous change in terminology during the recent years indicates the change of the perception of the software engineering process. Thus, the term *software engineering* is used today instead of *programming*, and 'software tools' are introduced adopting industrial production paradigms.

The following two subsections introduce briefly a terminology which seems to be common, at least to some extent, to all the environments that will be presented. Many of the terms are proposed and described in [13], but the list presented here, has been augmented with elements from all the studied articles.

All components, activities and functions described here, are present in all categories' environments, since all support the program implementation phase in one way or another. Various terms will be explained on the spot during the course of describing environments of the more advanced categories in later sections.

Programming Activities and Components

The main programming activities that a PSE is expected to support are the creation, modification, compilation, execution and debugging of programs. In the general case those activities apply to *software modules* instead of programs, where a module is a separately identifiable collection of information from a software database. The *database* serves as a repository of all information relevant to a programming project throughout its life-cycle. It may include programs in various forms and multiple versions, documentation, test data, analysis results, project management information, user manuals etc. Software engineering databases are also being used to provide an unifying and integrating medium for the tools in many environments. Tools can use the database to obtain information and return their results to it without having to interface with other tools, eliminating at the same time the need to keep multiple copies of the same information.

A way of grouping the modules in a database is to separate them into mutually exclusive sets called *partitions*, which can be used to distinguish different projects under development, different programmers, different system releases, different target configurations, and so on. A general study of the database structure needed for PSEs can be found in [80].

In the software module abstraction, each module has certain attributes like a name, a version designator, access control information, type, and so on. Some tools of the PSE may transform a module (or a set of modules) and produce a new module which is named and marked as *derived*, whereas the input module(s) is named *parent(s)*. A module that is not derived is termed *abstract module*. PSEs in most cases have tools to keep track of the *derivation history* coded in the *derivation designator* which contains the information to specify the parents of the module, the tool that made the transformation, or even the depth of a derived module from its most abstract predecessor. The derivation designator provides also a means of distinguishing a derived module from its parent.

During the life-cycle of a programming project, several *versions* are applied to its modules. Each revision results to a new version, that is a new module at the same level of derivation as its predecessor. A *version designator* is an attribute of a module which permits the PSE to order its several versions in time sequence of their creation. Version designators along with derivation designators may also be used in more efficient application of a transformation process, transforming only the modified parts instead of the whole module, provided that this process follows each revision. A mechanism for *stamping* every revision with its number, creation data, author and file name is usually provided for identification purposes. For the prevention of conflicts, a way of *locking* the modules is provided, feature that is very useful when more than one programmers work on the same modules (what is the case in PSEs). Another way of *protection*, at a lower level, is sometimes offered by the data abstraction mechanisms of the programming language itself (ex Ada[®], Modula-2). In most cases revision control systems store *deltas*, that is differences between successive revisions, resulting in considerable space savings, since less than 9% of the lines are changed from one version to the next. (see statistics on the SCCS and RCS in [81], and on DSEE in section 2.1.3, p 26)

Revisions are in some cases structured not in sequence but in trees (RCS) allowing experimental module modifications separate from the main-line development or even temporary fixes to a given release, while the next one is already being worked on.

Most PSEs provide some kind of *version control mechanism* to determine if a given module is up to date and insure that each module has a correct version designator when it is created. It is typical to 'freeze' a set of modules to form a *current release* and then continue revisions directed toward a new release.

Ada[®] is a registered trademark of the U.S. Government, AJPO

'secret' for integration. This methodology may have the form of common file naming conventions, database access techniques, command language or menu style, and system access - data structures

The environments of this category are distinct also due to the support they offer during verification, validation, and quality assurance phases. Verification and validation may be informal (e.g. *inspection*), semi-formal (*testing, symbolic execution*), or formal (*verification, static analysis*). Verification is usually applied to high level specification languages through *specification checkers*, and *incremental verification* is applied to LISP and LISP-like programming languages. Validation support is varying from *simple debuggers* to *behaviour analyzers*, and *structure analyzers*. Structure analyzers analyze module interrelation, and interface descriptions. Quality assurance and structural modelling prevents structural decay over the life time of the product.

An apparently contradictory property, that integrated environments should still have, is *granularity* of their tools, that is, tools should be small and orthogonal. Many of the well established tools (e.g. compilers) in the field may need to be finer grained, in order that parts of them to be used separately (e.g. the analysis component of a compiler). Granularity is being achieved, relatively easily, in many of the simpler environments of the previous category (e.g. UNIX), but in environments intended to accomplish much more complicated tasks using more sophisticated tools, the need for clean interfaces and simple functionality is emerging. Thus, languages like Modula-2 and Ada, that support system modularization and module specification/body separation, could help an environment to provide integration and granularity.

A general software engineering (or even engineering in general) principle, which is also found in many of the simple PSEs, applies of course to the IPSEs. It is the principle of *reusability*. Reusability in an IPSE means that the same tools can be used in several places helping to accomplish distinct tasks. This is *reusability of resources* or utilities

distributed model of computation as opposed to the block structured model (improvements in intermodule interface). Individual Workstations facilitate a lot the development process with their instant power and their advanced interfaces are going to change SEEs forever.

The majority of the SEEs are bound to some specific programming language. Software development seems to be more productive, effective, and complete when it is done in an environment especially designed to support just one language. When an environment happens to be implemented in the language it supports, it has the advantage to be *portable* and self maintainable (ex. APSE, Smalltalk, Cedar).

It is generally accepted that the language is one of the more serious prohibitive factors in the development of the illusive 'general purpose' SEE. Due to the great differences in programming philosophies, an environment would risk its consistency and effectiveness if it was to support languages fundamentally different (say Lisp and Modula-2). Nevertheless PWB and DSEE are environments totally language independent. Furthermore ARGUS is supposed to adopt the language of the host machine. Since ARGUS will provide mechanisms for the selection of facilities depending on the project purposes, it can be considered as a *meta-SEE*.

As regards to the requirements specification and design phases of development, there exist methodologies and environments who support them, that are completely independent of any programming language. The reason for this independence is the use of various high level program description languages (PDL), the use of pseudo-code, or various design notations, like data flow and structured design charts (Yourdon [88]), program flowcharts (Nassi-Shneiderman [59]), data structure notations (Jackson [41]), logical structure notations (SWDA [4]), and Petri-nets for the representation of systems with concurrency or parallelism.

New languages that will allow dynamic addition of processes will eliminate the need

to shut down a system to reload a program, giving more power to the host, in host/target oriented SEEs. NIL [16], [73] is a language which provides this feature. A whole software methodology is based on the language. This methodology emphasizes portability and extensibility, combining new powerful features like exclusive process ownership. System configuration logic is included, for the first time, in its modules rather than in external binders and linkers.

The following four subsections present many characteristic SEEs, exploring their features. Each one subsection deals with environments from one of the already defined categories.

2.1.2 Programming Support Environments

Early software programming environments consisted of an editor, a compiler, and a linking loader. Later environments included debuggers, source code and text managers and informal requirements and design methods. All those tools were added as independent, diverse components and do not form integrated environments. It was estimated that three years ago there were more than 400 commercial software development tools existed. [40].

The simple interconnection of any set of those tools does not produce an integrated environment in any sense. It results only into a complex structure with no consistent set of abstractions. The existence of a common kind of data object, used by every single tool or facility in the environment, is the only possible way to provide integration. This helps the users to form in mind a concrete conceptual model of the environment, and makes their work easier and more productive. For instance Interlisp is built around the use of list structures (S expressions) as the basic data object, and UNIXTM uses the file as the common building block.

UNIXTM is a Trademark of Bell Laboratories.

Elementary PSEs

The number of people who see the UNIX not as an operating system but more as an operating environment is growing every day [83]. UNIX was expected to become the minicomputer industry's single operating system but might still become the standard PSE, while nowadays the term 'operating system' is becoming obsolete. The UNIX operating system provides a PSE based around the idea of the toolset. This PSE gives a large degree of flexibility and ease of accomplishing programming tasks.

The UNIX programming philosophy is based on the relationships between programs. To use the system well the programmer has to understand the tools' functionality and how they fit into the environment. The *UNIX shell* is the interactive command interpreter and at the same time the programming language, shaped as much by history as by design. Its drawback is its unusual syntax, but its power lies in the fact that it is a high level, interactive program development tool. Its approach is a general characteristic of UNIX: build on what others have done instead of reinventing the wheel every time, and use the tools to experiment with new ideas [43].

Some of the tools facilitating program development in UNIX are the *make*, a program for specifying and controlling the processes by which a complicated program is compiled, the *yacc*, a parser generator (generates a parser from a grammatical description of a language), and the *lex* for lexical analysis. For large program maintenance in a production environment the Source Code Control System (SCCS), and the Revision Control System (RCS), are offered. In this case (programming in the large) the Programmer's Workbench (PWB/UNIX [53]) system provides a front end system that allows programmers to use the UNIX system and a number of tools to develop and administer non-UNIX, target independent code. PWB provides facilities for documentation, for managing a project database, for maintenance, and for communication within the programmers team.

Mitze [53] reports on two software development projects using UNIX, on the problems

that non-experienced users encountered, and on the C global symbols problem. The latter is caused by the fact that C symbols must be declared before their use, and since in large projects symbols are frequently defined in terms of other symbols with the definitions in separate header files, there is the possibility of circular deadlock. Data encapsulation capabilities, found in new languages is one solution, and tools that will take care of setting up include file hierarchies is another.

UNIX of course doesn't define any simple software development methodology. Projects have to use their own methodology depending on their needs. A set of nice tools, and an excellent environment for developing new tools is what UNIX system provides.

INTERLISP, 78 is a pioneer programming support environment, that 'evolved' for LISP. It grows as a collection of tools to facilitate LISP code programming and debugging. It has been used for many years with remarkable success, aiming in the production of complex software, allowing *interaction* and *incremental programming*. What must be considered is the contribution of LISP to the growth of INTERLISP as a PSE. LISP offers unique data structuring facilities, it is extensible and incremental, and has internal representation of programs. The same is not true for the more conventional languages, which need more sophisticated program manipulation systems.

Many of the features that INTERLISP introduced for the first time, have become a standard for today's environments. *DWIM* (Do What I Mean) is a spelling corrector facility at the system command level, which is invoked when the system detects an error and attempts to guess what the user may have intended. Facilities like the *UNDO* and the *structured editors* are also INTERLISP's contributions to the improvement of PSEs.

Another PSE, which introduced windows, menus, and icons to the world of personal computers for small program development, is the *Smalltalk* environment. 29 It is built around Smalltalk-80 which is an object-oriented (non-procedural) language. The most interesting point here is the absence of operating system. Everything is within the

language

A completely different approach is that of APSE (Ada PSE) [9]. It is a *layer-approach*, having the significant advantage of portability. An Ada environment consists of

- *Kernel APSE (KAPSE)* Provides runtime support, communication and database functions, and serves as an interface between the higher levels of APSE that are written in Ada and each individual machine's software. Ada intends to provide standardization by specifying the outer interface of the non-portable O/S facilities.
- *Minimal APSE (MAPSE)* It is the minimal Ada programming environment. Consists of a minimal set of tools (editor, compiler, interpreter, debugger, linker/loader, and configuration manager), all written in Ada and supported by KAPSE.
- *APSEs* Environments bound to particular applications or methodologies, constructed from extensions of MAPSE.

Gandalf [51, 32] is an implementation of APSE. It provides support tools for system composition and generation, incremental program construction, and project management.

Gandalf offers a *structure directed editor*, the ALOE, which provides the user with templates of language constructs to be filled up. It is possible for the user to write his own parser and to use it for limited text editing. The System Version Control Environment (SVCE [33]) supports system composition and generation providing facilities for system design, intercomponent connections, and automatic multiple version control and generation. Gandalf is directed towards programming in the large and has not, until recently, been targeted for personal machines with graphics capabilities.

The CNET project [54] offers a specification for an APSE implementation. It is directed towards the development of functionally distributed systems.

Lately there is a growing negative reaction to the APSE attempt for standardization on a uniform set of internal system interfaces and on a single programming language. [85]

Elaborate PSEs

Programming-in-the-large, that is developing systems consisting of thousands modules with maintenance lifetimes of many years, and a staff of many people, is of course a complex process. Environments that will be reported from now on are supposed to use computer systems to automate many phases of this process. They differ from the elementary ones, in that they support project management in addition to the program development phases.

A description of the methods used by two big software houses in defining and implementing software development environments is given in [21] and [20], and it is representative of the way industry and government face the software development process. The major objective there, is to have something up and running in short time after the start of a project, in order to justify any corporate investments with the analysis of its effects. A software office of the future was defined to achieve productivity improvement, (increase by a factor of four by 1990), where software tools were the critical part of the strategy [21]. The result of this definition was the establishment of a Software Productivity Project (SPP) with a team of about 20 experienced people, several VAX 11 780 in a LAN running Berkeley UNIX. Among the ten tools that were chosen based on project restrictions are a Program Design Language, a Relational Database Access to modify a centralized relational database (the INGRES), a Software Requirements Engineering Methodology for defining and analyzing software requirements, a Fortran 77 Analyzer for automated analysis of programs, an Automated test manager to support the testing of Fortran software units and the Berkeley UNIX facilities, namely the standard UNIX

features together with a distributed architecture. Most of those tools are independent, unaware of the existence of the others. A big part of the work has to be done by analyzing and understanding the interrelationships between them, in order to transform them to an integrated system, a task, which may not even be possible.

Another similar, proprietary, software development system consisting of integrated development techniques over the life cycle, a set of software tools and a physical facility for the development and test of software, is described briefly in [20]. The number and the kind of tools used is almost the same as in the above described environment with emphasis on the validation and verification in each life cycle towards an early detection of errors. The observations that the cost of rectifying an error later in the life-cycle is, by a factor of five or more, higher than if detected in the early construction and integration stages, led them in the introduction of automatic test tools. An Ada PDL processor was used to verify interface relationships among the various modules by automatically checking Ada package specifications.

2.1.3 Integrated PSEs

It has been proved that simply a set of tools is of little use when we have to deal with big and complicated software projects, and the word *integration* became part of our terminology. In a very loose sense, even programs that can be executed on any operating system can be termed integrated. But in the case of software engineering environments integration of this kind is in most cases useless. The fundamental way in which a toolset is integrated, is the sharing of utilities (database access, revision control, statistics and accounting mechanisms), and the existence of common kind of abstraction internal to the tools. The goal is to make the PSE to appear like a single tool, a tool that does not require the programmer to perform mental context switches.

The existence of an overall methodology which will make the environment seem 'usage' oriented, and will be used for the realization of all the partial tools, it is the

'secret' for integration. This methodology may have the form of common file naming conventions, database access techniques, command language or menu style, and system access - data structures.

The environments of this category are distinct also due to the support they offer during verification, validation, and quality assurance phases. Verification and validation may be informal (e.g. *inspection*), semi-formal (*testing, symbolic execution*), or formal (*verification, static analysis*). Verification is usually applied to high level specification languages through *specification checkers*, and *incremental verification* is applied to LISP and LISP-like programming languages. Validation support is varying from *simple debuggers* to *behaviour analyzers*, and *structure analyzers*. Structure analyzers analyze module interrelation, and interface descriptions. Quality assurance and structural modelling prevents structural decay over the life time of the product.

An apparently contradictory property, that integrated environments should still have, is *granularity* of their tools, that is, tools should be small and orthogonal. Many of the well established tools (e.g. compilers) in the field may need to be finer grained, in order that parts of them to be used separately (e.g. the analysis component of a compiler). Granularity is being achieved, relatively easily, in many of the simpler environments of the previous category (e.g. UNIX), but in environments intended to accomplish much more complicated tasks using more sophisticated tools, the need for clean interfaces and simple functionality is emerging. Thus, languages like Modula-2 and Ada, that support system modularization and module specification/body separation, could help an environment to provide integration and granularity.

A general software engineering (or even engineering in general) principle, which is also found in many of the simple PSEs, applies of course to the IPSEs. It is the principle of *reusability*. Reusability in an IPSE means that the same tools can be used in several places helping to accomplish distinct tasks. This is *reusability of resources* or utilities

(software components), in performing a variety of tasks during the software production process [85, p.13]

Application generators support the production of granular software components and also the environment where they can be used efficiently, both at the same time. This is accomplished by taking advantage of the reusability of the mechanism for generating software components as well.

PDS (Program Development System) [11], [12] is a PSE that supports the validation phase in addition to the main programming phase. It is developed around EL1, a LISP-like language. It provides a program database, a version control mechanism, a command interpreter and a set of tools. The integration of tools is achieved by using a common internal representation of programs, by sharing the database, and by allowing mutual invocation of tools.

The Cedar Programming Environment [79], is an interactive, experimental programming environment, that features a unique highly visual user interface which exploits the high bandwidth display and mouse device of the Dorado Xerox PC. It provides uniform screen paradigm using the *Viewers Window* package with icons, whiteboards and various tools, all written in Cedar. The Cedar programming language is a strongly typed, compiler-oriented language of the Pascal family, based on Mesa. The fact that the Cedar language is based on Mesa does not prevent the Cedar environment from supporting Interlisp and Smalltalk programming styles. For example, the Cedar interpreter is an important and integral part of the Cedar environment, greatly facilitating, debugging and testing, even when the programs being debugged are themselves totally compiled. The important point here, is *the ability to create and execute program fragments in a specified dynamic context*. Interlisp provides this ability using a separate interpreter whereas Smalltalk compiles each expression.

Abstraction mechanisms help keeping development, debugging, testing, documenta-

tion and maintenance of programs in manageable pieces, giving the programmers the freedom to design those pieces.

The Cedar user interface allows individual interactive programs to operate in a non-intrusive manner with respect to the user's activities. (*Principle of non-preemption*) Background tasks run during the time the user is thinking or the time between keystrokes. The use of several *Work Areas*, that is instances of *User Executive* associated with a viewer provides parallelism. The user works in several work areas at the same time having initiated several instances of the executive, each with its own state and performing its own operations. Tools like the 'watch' or the 'Cedar Spy' can be used by the programmer for *performance monitoring* during the program's execution. The Spy tool provides measurements of CPU cycle consumption, page fault and procedure calls. Furthermore Spy can tell where a process is spending its time, where is waiting on monitor locks, where is waiting on condition variables, and when is preempted by other processes.

Most of the capabilities of the Cedar IPSE were first defined in the 'Requirements for an Experimental Programming Environment' Xerox PARC report [19]. It is in this report where are gathered the *principle of non-preemption*, the *law of least Astonishment*, the need for integration with the coexistence of the various facilities in the same address space (no explicit context switches), the observation that programming should be *think bound*, not *compute bound*, leading to fast turnaround for minor program changes (<5 sec), the principle of consistent user interface, and many others, all undoubtedly based in careful in depth study of the state of the art.

The DOMAIN Software Engineering Environment (DSEE) [47] incorporates some of the best ideas from existing systems in a single program unit, with instances running at various nodes in the Apollo DOMAIN network. DSEE is a distributed, production quality SEE, for large scale development projects. It consists of several 'managers'.

- History manager, to control source code and provide version support.

- Configuration manager aiming in building a system from its components, and detecting the need for rebuild
- Task manager, to correlate source code changes made throughout the network to particular high level activities (tasks)
- Monitor manager, to watch user defined dependencies, and alert users when such dependencies are triggered
- Advice manager to hold general project information, and provide templates for redoing common tasks.

Due to the limitation in space, only the History manager is chosen to be presented here.

DSEE uses a distributed database management system (D3M) to store historical information, which is manipulated by the *History Manager*(HM). The HM is a *reserve replace* and incremental change (*delta*) oriented system. Related source modules (*elements*) are grouped into *libraries*, in highly-compressed format. Users *reserve* an element for modification and edit a local copy. When they have finished changing and testing the copy, *replace* the element, creating a new version. The HM inquires about the reasons behind a change, and then records that information, along with the date/time, node id, and person's name in the history database associated with the library. Of course the system takes care of conflicts and informs users when an element is used, stating why is being reserved and by whom. Further more, the different instances of DSEE managers are informed when a new version is created using inter-process communication messages.

DSEE, as it is been said, stores only deltas between successive versions (like RCS). Worth noting are the statistics that are given in [47, p.106]. According to those on typical Pascal modules, each new version makes the delta file about 1%-2% larger, that is to say 50-100 versions of a module can be stored in the same space as 2 copies of that module. In addition to this, with the leading blanks' compression method that DSEE

uses (space count bytes), an HM element with 5-10 versions is often smaller than a single clear copy of that Pascal source element.

DSEE is a host/target oriented system. It allows for alternate, target-dependent implementations of the same module, while providing means for development

Another host/target development environment is the DICE [25]. Moreover in a Distributed Incremental Compiling Environment host and target are integrated, the host having complete control over the target. It is able to stop and start execution on the target machine and also manipulate code and data inside the target. The prototype version of DICE is implemented in INTERLISP, in order to be incremental and interactive, and supports the development of large PASCAL programs. Its main features are:

- Statement level incremental compilation
- Cross-reference information is stored as a part of the symbol table. DICE maintains a cross-reference and static-analysis database
- DICE language oriented editor is not a pure tree editor (like in GANDALF), but a hybrid text and structure editor for maximum flexibility
- Remote debugging and maintenance.
- Separability, since the compiled program is separated from the source code, so that, it can execute outside the program development system.
- Connectivity. DICE can be connected to a malfunctioning production system in order to debug it or to correct it. This can be done after the error has occurred and does not need to be planned in advance.
- DICE prototype version is not portable, since it is not implemented in the language it supports. It has been decided though the application of a *bootstrap process* to

produce a new portable version, an IPSE which can be maintained by itself. The strategy is called *prototype bootstrapping*, and is analyzed in [25].

An IPSE that is intended to prove, like Cedar, that environments based on *incremental compilation* have the ability to specify both execution and debugging actions in the source language, exactly like the *interpretive systems* (INTERLISP, DICE), is the Magpie [18]. Magpie is an interactive Pascal IPSE. It supports immediate execution of Pascal statements and also user-defined Pascal procedure to monitor events. It provides a nice, consistent user interface with *browsers* (windows), pop-up menus, the same arguments for all commands, and language directed editing. All those features together make Magpie easier and more pleasant to use while abolish the boundaries between programmers and debugging.

A completely different approach is offered by PECAN [66], which is a family of program development systems. These systems use the concept of multiple views of shared data structures. PECAN is currently being implemented on Apollo Workstations. Many of its main features are found in some of the above mentioned IPSEs and these are:

- Immediate feedback of semantic and syntactic errors while the user is editing
- Undo/Redo facilities.
- Structured templates for building the program are available as commands. The flexibility to type text at any time instead of using templates.
- Use of on-screen and pop-up menus
- Multiwindow display.
- Incremental semantics that allow the program to be compiled as it is edited
- A framework that handles a variety of algebraic programming languages with the same commands.

Since different users prefer different languages, the goal of PECAN designers is to be able to generate a PECAN environment in about a week time[66, p.327]. The PECAN philosophy is that of layer-oriented systems like APSE, and consists of an hierarchy of four layers built upon the UNIX operating system.

A development system for a broad range of distributed microprocessor architectures and applications is the MML [3]. It is written in Pascal, runs on a PDP and produces executable programs for a Z8000-based multiprocessor system. It provides debugging facilities but the host/target system is not completely integrated, like in systems for single processor targets (ex. DICE).

2.1.4 Design Support Environments

In this subsection environments that support mainly the design phase of the software life-cycle model are presented. They may also support some parts of the implementation and specification processes, but this is done independently and there is no direct relation between the tools, much more there is no provision for automatic transformation from one phase to the other.

The need for design and analysis at a higher level than programming led to the evolution of DSEs. These environments take advantage of the intuitive aspects of iconic communication in describing design, control, data, and concurrency issues. The design is represented using either some kind of formal description language (e.g. BNF), or graphics. The graphical representation of design's structure is generally accomplished using some set of distinct geometric shapes, to represent and distinguish entity attributes (e.g. types of modules), and directed arrow connections for entity relationships (e.g. calls). The representation of a system's temporal behaviour is achieved sometimes with the use of another kind of graphics symbols, either of the Petri-net form, or SDL (Specification and Description Language). When the design process takes place on a graphics workstation interactively, the ease of exploration and comparison of alternatives is one of the

most attractive features of the environment.

Some sort of logical representation of the picture is of course needed, if the DSE is to be able to make any kind of useful reasoning about the designs. Thus, a logical structure manager must exist in conjunction with the picture manager. Among the responsibilities of the logical structure manager is the syntactical correctness checking, the consistency checking, and the verification of correctness of any changes in the design.

Many of the DSEs had been developed to support handling of structure charts of constantly increasing complexity, like AIDES and ARGUS. Other DSEs came as a solution to the problem of designing complex non-sequential computer systems, like ISES and DREAM. Even more, some others constitute front-end design systems producing output that forms the basis for the automatic production of compilable HLL code (CAEDE, MELBA).

The Automated Interactive Design and Evaluation System (AIDES), [87] which has its roots in the structured design methodology and in some of the first software engineering environments, and is directed towards large systems design. It supports mainly the design phase with graphics, and quality measurement, and provides some tools for the requirements and the programming phases. Needless to say that all information transformation from one phase to the other, is being done manually, and the only unifying force between the tools of the environment is a data base (ADBMS) for the store and retrieval of modules and documentation. The design notation is that of structured design and the tool used is the old Structure Chart Graphics system (SCG, 1977). Worth noting are some functions provided during the design phase. The user has the ability to view the concept of structure (zoom), and the layout of the structure is automatic. Measurement of design is automatic too, with reference to complexity, independence and infinite loop recognition.

PegaSys [55] and [56] helps expressing a program design as a hierarchically structured

tiprocessing, it has complete monitoring capabilities over the system that is connected to, and it is especially well suited for host-target development environments. For these reasons it was used as the basic building block in the construction of the herein presented prototype DPM¹

2.3.3 Real-time Embedded Systems

The development process for these systems takes place on a host computer where the SEE resides. This is because embedded systems cannot afford to provide their own development environment, being usually very small systems. During the last development stages, target code is produced using cross compilers, cross assemblers, link-loaders and it is downloaded to the target.

Initially only the target hardware is available, and after the development and download of the first prototype, both debugging and performance monitoring processes can only be performed on some software units. The use of an environment simulator hooked to the target can be sound the start of the performance measurement of the integrated system. However, it is only when the environment simulator is replaced by the real world, that the full system performance measurement and analysis is begun.

Apparently integrated performance monitoring systems have only lately start been used on target computer software. This problem area is seldom considered in most of the literature on tools and environments. Virtually, no high level tools are available in any aspect to support the real-time software performance monitoring, in such a way that the designer gets a clear picture of system's performance during execution, without the need to be aware of the system's implementation details, at assembly language level.

There are several difficulties related to execution monitoring of real-time programs, imposed by the nature of real-time processes and their concurrency.

¹Up to date no single device with such capabilities exists in the market. Some companies though, are developing new products to fill this gap.

science training, but facilitates a lot the conceptualization of concurrent software systems during their design.

In the appendix of [68] there is the DDN description of an operating system (similar to 'THE operating system' by Dijkstra). A quick study of this sixteen pages long, high level code (pseudocode plus DDN constructs), proves once more how much difficult is the design of big systems without graphical notation (DDN's graphics are rudimentary). The comparison of parts of the spooling system's design with a *spooler design* given in [5] which is done using SDWA notation [4], shows how more complicated to use is any formal description language versus a graphical notation language, during the system's logical design process. Of course we do not make any efficiency comparisons at this point.

Every DSE is designed and built in order to support a specific methodology. So, design methods being diverse and unrelated, form yet another obstacle in the development of the 'general purpose' SEE.

2.1.5 Software Development Environments

It starts becoming understood that Software development needs both graphical and textual representation. IPSEs do not support design and DSEs do not support integration between design, graphics and programming.

Maintenance in SDEs is also emphasized, and is being considered to be an integral part of the design, or even the requirement analysis process, subject to the same systematic treatment as the rest of the development process.

Arcturus [77] is basically an APSE implementation but it is presented in this section, due to the fact that provides support for the design phase using Ada PDL, and automated stepwise refinement from Ada program designs into executable Ada. Furthermore it offers an approach to the integrated use of compiled and interpreted Ada, and performance measurement with colour profiles. The current implementation is written in C and runs

on a VAX under UNIX, on top of the Ada Virtual Operating System.

CADAS [8] is intended to provide APSE implementations with support for prototyping, verification, and validation, transforming them into complete SDEs. Its application area is the embedded systems area, where its mechanism for providing a rapid prototype (a 'software breadboard'), is best suited. The key features of CADAS are for prototyping :

- the cross referencing of the prototype specifications to the requirements
- the provision of an events language
- the capability to generate and insert models automatically into CADAS

for testing, verification, and validation :

- the ability to perform RT closed loop testing
- the exercising of the preferred models of operation in RT
- the identification of the exceptions within a system with the appropriate elements of the environment
- the exercising of the identified exceptions.

ARGUS [74] is an SDE which provides tools for interactive graphical design (STADA tool) to support structured design, a dynamic analyzer (DYNA) to observe and document software behaviour during execution, and an architectural analyzer (ARCA) to analyze system structures such as interface descriptions and module consistency. The existing implementation is used for the development of Fortran programs.

There are significant differences between parallel programming in multitasking or multiprocessing systems and sequential programming so fundamental that the development of parallel systems has to be done on environments specially designed for this

purpose. Prometheus [2] is one such environment designed to support the production of parallel programs with interactive description refinement and program editing. It is planned to offer three different execution sub-environments for the testing phase supported with an execution monitor and a graphical debugger. The monitor will maintain execution path histories and the debugger will interrogate histories in a post-mortem analysis, being able to distinguish among parallel control paths and concentrate on a certain program functions subset during interactive debug mode.

There are SDEs which provide for the automatic transformation of symbolic information from the specification-design levels to HLL code. These are systems which either

- use a kind of abstract representation of the structure of the designed system, information on its temporal behaviour and its data structures or ADTs, to produce compilable code,
- or combine structure representation with incerted code segments,
- or produce programs by linking predefined sets of subroutines together. This is the 'reusable code' approach to rapid prototyping, and is used in business applications usually. It is analogous with the use of integrated circuits in the hardware area.

Only systems using the former method are HLL independent and of general interest. One of the first projects in the area of automatic code generation was the SX1 of British Telecom which used a textual form of input to produce SDL-like diagrams and PRO-CORAL code. Nevertheless, the software designer had to include target language statements in the graphical symbols.

MELBA [23] is one of the least automatic code generation environments that are completely programming language independent. It uses graphics Module Structure Diagrams to specify structural behaviour, and SDL (CCITT specs) to specify temporal behaviour and libraries of data structures. It produces compilable CHLL code for com-

munications systems, but it is not limited to that at all.

Up to now no SEE exists, which integrates concepts, methods, models, and tools for the entire software development process. Characteristic of the state of the art in this area is the diversity of view. Comparison of the referenced environments was not easy, even after the proposed classification, mainly because of the distinct features that are provided, and the lack of agreement upon terminology.

2.2 Rapid Prototyping

Prototyping has been an important part of the development process of any new product in most branches of engineering. A prototype is built to assess the system before the final design. If the prototype fails to meet both the functional and nonfunctional specifications, it is modified and reevaluated before the product construction permitted to begin. In software engineering though, things are different, since there is no production of large number of product copies (production run), and the prototype itself, is the product. There is no mass production to cover the cost of prototype, and therefore it is argued that prototypes are cost defective.

Prototyping means building an application while feasibility of the design remains in question. Therefore prototyping is basically a feasibility study that serves to demonstrate system aspects critical to the user [63].

In the conventional model the initial phases of requirement specifications and analysis is a slow long-lasting process. The implementation of the application system is delayed and little or no feedback is permitted from the end user, until the final development phase, the *user acceptance test*. If it happens to be proved that the system does not meet user's expectations, reexamination of the specifications, the design, and the code becomes a very expensive process, constituting another phase, which after all, is not included in the model. Another major disadvantage is that there is no such thing as

software reusability. All the application software is designed and implemented every time from the scratch and no existing software is used.

In *Rapid Prototyping* the specifications are not completely defined, but are subject to revisions and iterations. In the way the concept is used in this thesis, the design is not finally specified, when the system's early implementations execute on the target. Reusability is applied in large degrees, and the system is in existence much earlier, exposed to criticisms and improvements.

2.3 Performance Measurement and Evaluation

Some of the main concerns during the course of the thesis were the ones related to performance measurement, monitoring, analysis, and evaluation. System performance was considered to be of great value, since it has been proved that high performance systems can only be produced, if performance had been studied since the early development stages, and then tested after the implementation. Generally, the system performance measures fall into the following categories

Throughput A measure of the computational work accomplished by the system per unit time. If n jobs are completed in an interval of t seconds, the throughput is taken as n/t jobs per second during that interval. No generally acceptable definition of a unit of computational work exists though. System throughput depends on system capacity, the time interval over which throughput is measured, the load on the system (workload characterization problem), the scheduling method and the job mix. An analysis of the system with respect to these factors needs to be conducted, for the throughput value to be of any meaning. Moreover, considering throughput alone as a figure of merit for a system, can prove to be a rather deceptive measure. For example, systems giving high priority to small jobs are achieving high throughput but also a backlog of long jobs. Especially in the case of real-time embedded systems, which is of special interest here,

this measure is of minor importance and consequently is going to be considered as such.

Responsiveness : The response time to interactive commands and job turnaround time, are typical measures of responsiveness. For real-time systems, the latency (response time to interrupts), service times and messaging delay times are crucial measures, often of higher priority than any other. Tasking and real-time issues though, cannot be considered by their own. Development and analysis of multitasking programs involves looking both at the tasking and sequential aspects of the program.

The main objectives for the measurement and evaluation of a given system are:

- Aid in the design and development of better hardware and software.
- Aid in the selection of the best suited system and evaluation of any changes in configuration.
- Help in improving the performance of an existing system (Optimization, Tune up)
- Evaluate software design alternatives either using simulation or by means of an on-line dynamic performance monitoring and analysis system. (Present case)

Various tools and methodologies exist to support systems' measurement and evaluation. Figure 2.1 illustrates the way they can be classified, and in the following sections, these methods are analyzed.

2.3.1 Performance Modelling

The use of models is needed when actual measurements cannot be made, or in order to predict the performance of a not yet implemented system [76] [44]. *System models* include functional (Petri-nets, flow charts), simulation, and stochastic models, and describe the relationship between workload measures and performance measures. *Workload models* can be either executable (benchmark set of real or synthetic programs), or nonexecutable, used in conjunction with a simulation or analytical system model. The use of any kind

Figure 2.1: Performance Tools classification

of model either of system or workload results in performance prediction, very dependent on the model. When analytical models are used, accuracy need not have statistical dependence, whereas for simulation, accuracy depends on the run-time of simulation, and results are not valid if steady state not achieved.

Even though performance modelling is not of prime interest in the contacted experimental work, p-model, the tool that is already part of CAEDE, it is intended to be used in conjunction with the monitoring tool (DPM) during development, where DPM provides another important function, the validation of the results from the performance prediction phase

2.3.2 Performance Monitoring

When the system is functioning, *monitoring* could take place instead of modelling to measure performance. In this case, the measurements are only dependent on the work load during testing, factor that does not affect the validity of the results, if one suppose that the load represents exactly the real world situation, as in the case of systems examined here.

The main functions of any monitor are event detection, data collection, data reduction and presentation of results (Fig. 2.2) [22]

Figure 2.2 Performance monitoring system functions

The detection of a certain event causes the monitor to collect and analyze an amount of data that are accessible at this specific moment by its instrumentation. The types of events to be detected, the amount of collected data, and the degree of their immediate reduction vary with the type of monitor and may be definable by the user. Basic types of *events* are the execution of a certain program instruction, the access of certain memory locations, the updating of a register and the hardware or software interrupts. Event detection is accomplished either by the insertion of a special instruction into the program at a certain location (*software probe*), or by using hardware circuitry to monitor the system address bus and control lines. This kind of monitors can be termed *event driven*.

There exist *sampling monitors* as well, which either use an interval timer to interrupt the CPU and take control over the system, or again use special hardware to monitor and randomly sample every aspect of the system during execution, constantly altering state between data collection and reduction. This continuous state switching is needed because monitors of this kind are trying to balance between *tracing*, that is minimum or no loss of information, no immediate processing, but maximum storage requirements, and *counting*, that is maximum immediate processing, minimum storage but maximum loss of information. In general, sampling monitors are not as accurate as the event-driven ones, for certain types of indices, and they sometimes have limitations as regards to the collection of protected system information (software monitors), or data related to sporadic events (hardware ones).

Depending on the delay time between the extraction and analysis phases, a monitor can be considered as *off-line* or *on-line*. If the system supports only data acquisition at run-time, and differs the presentation of the results or even the final reduction of data until later time, it is an off-line monitoring system. When analysis is performed on-line with respect to extraction the system is termed on-line monitor.

System-oriented monitors usually measure system performance indices, like response times, throughput rates, and resource utilization, as well as system and workload variables, like CPU time demands, memory space demands and degree of multiprogramming and parallelism. *Program-oriented* monitors intend to measure program and module execution times, total CPU times, instruction execution counts, and frequencies, constructing the profile of the executing program. An example of program-oriented monitor is the profile option of the C compiler under UNIX.

Instrumentation methods

There are two kinds of monitoring systems as regards to the instrumentation for event detection and data collection functions :

Software monitors are pieces of software inserted at compilation time in the application, or long before application development in the operating system itself, in order to collect performance data. They can be *permanent* or *temporary*. Permanent instrumentation is incorporated into the system for its lifetime and therefore its interference (although not the system overhead they produce), can be ignored. A system-oriented permanent monitor exists in UNIX, keeping statistics of the system usage by the various processes, and taking advantage from the fact that an application program is a system process, it can give application accounting too, with the provision of the *times* command.

Temporary instrumentation is being used for short-term system tuning, real-time

detection of performance problems, and may be for continuous or periodical monitoring according to a performance management plan, whereas permanent instrumentation is best suited for the latter. Temporary monitors alter the measured quantities in a predictable way, interference that cannot be ignored, especially in real-time systems, where a disruption of time relationships is unacceptable

Hardware monitors are usually devices independent of the system under development, which 'listen', to the transactions occurring on the system. Since they do not add any software load to the system neither change the execution in any way are termed *passive* monitoring systems. Most of them are external general-purpose devices, but there are some that are designed and built integrated with the system they intend to monitor

Generally, a hardware monitor consists of comparators and logic circuits, counters and timers, a clock, data registers, and means of storing the collected data. It is connected to the processor control, address, and data lines, the system bus or peripheral controllers, via *probes*, that is, sets of signal sensors, designed for minimum interference with the system under measurement. Even though a direct measurement of the CPU busy time, I/O or disks utilization is achievable using this kind of hardware, only the use of software, either inserted in the application (software instrumentation to identify different types of events), or even better executing on the monitoring device, can give more vital information on system's performance

Current development in the hardware monitors shows a trend towards supporting the final data reduction and presentation of the results function, using microprocessors and special software support. Logic analyzers are a form of monitor to support microprocessor systems, even though they are primarily used for debugging (diagnosis) purposes. A special case is the Tektronix 1240 analyzer, which providing special software in a ROM pack to measure and analyze application performance, constitutes an on-line, passive monitoring system. Even though this package does not support multitasking or mul-

tiprocessing, it has complete monitoring capabilities over the system that is connected to, and it is especially well suited for host-target development environments. For these reasons it was used as the basic building block in the construction of the herein presented prototype DPM¹

2.3.3 Real-time Embedded Systems

The development process for these systems takes place on a host computer where the SEE resides. This is because embedded systems cannot afford to provide their own development environment, being usually very small systems. During the last development stages, target code is produced using cross compilers, cross assemblers, link-loaders and it is downloaded to the target.

Initially only the target hardware is available, and after the development and download of the first prototype, both debugging and performance monitoring processes can only be performed on some software units. The use of an environment simulator hooked to the target can be sound the start of the performance measurement of the integrated system. However, it is only when the environment simulator is replaced by the real world, that the full system performance measurement and analysis is begun.

Apparently integrated performance monitoring systems have only lately start been used on target computer software. This problem area is seldom considered in most of the literature on tools and environments. Virtually, no high level tools are available in any aspect to support the real-time software performance monitoring, in such a way that the designer gets a clear picture of system's performance during execution, without the need to be aware of the system's implementation details, at assembly language level.

There are several difficulties related to execution monitoring of real-time programs, imposed by the nature of real-time processes and their concurrency

¹Up to date no single device with such capabilities exists in the market. Some companies though, are developing new products to fill this gap.

Real-time processes are processes² whose correctness depends on their behaviour with respect to time (i.e. time in the sense of real-world time). They are used to control real-world processes, or analyze data generated by them. A process is not repeatable, since it incorporates time as a parameter. Therefore, new ways of monitoring have to be devised to cover the case of non-sequential, multiple state applications.

Observing the sequence of states of a process and assigning a true value to each of these, is the first stage of such a technique [62]. How this true value is to be evaluated is specified by means of a predicate. A positive evaluation of a predicate initiates an action, which can be used to record information about the process.

Usually an operating system keeps track, in one way or another, of the active process at any given instance. Measurement of the exact relative or absolute processor time consumption by the various processes can be achieved then relatively easily. This is the approach of DPM version 2.0, which will be presented in a following section.

A survey of the major quantitative methods used in computer performance evaluation, with a very clear classification of more than two hundred recent publications is offered in [38]. Despite the fact that in the part of this paper devoted to performance measurement numerous publications are cited, there is not a single reference related to real-time embedded systems. There are only two references to multiprocessor environments, those of Cm* [84], and Multics of MIT [69]. The latter representing an old architecture and the first having a very special interconnection topology. The work done for the performance analysis of the Cm* multiprocessor which is a powerful multiprocessor and uses a special topology in the processor interconnections, is very much 'prediction model-oriented'. A distinction has been made between microparallelism which is associated with a group of tasks highly interconnected performing control of one particular part of the system that is defined as a job, and macroparallelism, which is associated

²The terms *process* and *task* are used interchangeably herein to denote any 'dispatchable' unit, or concurrent module.

with the whole system. Prediction of the 'worst case' system performance, and the cycle processing power method versus the effective processing power of bus oriented multiprocessor systems is analyzed. One of the conclusions is that in most cases some maximal cycle time will be the most important factor in multiprocessor real-time systems since a system with just one iteration longer than maximal allowed could lead to fatal system behaviour, despite the fact that effective processing power could be satisfactory for system throughput.

Real-time systems software is by nature concurrent, a fact which imposes major difficulties in making these systems reliable. A tiny programming mistake can make such a concurrent system behave in an unpredictable, erratic manner. Classical debugging methods are not adequate for coping with tasking errors, either *deadness* or *sequencing*. Moreover, once the program is running, it is almost impossible to measure its performance with conventional hardware or software instrumentation. Detection of errors and gathering of runtime information must be automated.

Figure 3.4: The idealized, CAEDE based SDE.

In designing very complex software systems, there will be many places where the designer must choose among competing methods or algorithms.

Each of these choices will have some impact on the overall performance of the system, but it is nearly impossible to predict the ramifications of all these choices together. Again prototyping provides some help. *If prototypes of subsystems can be rapidly produced, performance measurements can be made during design.* If performance problems are detected, the design can be modified. Iterations of this process can help guarantee that the final design will lead to an implementation that meets the performance specifications.

[86, p.155]

Figure 3.1 shows the original waterfall life cycle model (W.W.Royce : 'Managing the development of Large Software systems : Concepts and Techniques', *Wescon Proc.*, Aug. 1970). In Figure 3.2 the herein proposed use of the rapid prototyping - automatic implementation, and performance monitoring - is shown.

3.2 An Idealized Software Development Environment

CAEDE [6] is an experimental design environment, which supports a structured design methodology (SDWA [4]), an iconic design entry system (using SUNcore graphics), a design database, and a set of design tools. It supports the development of complicated non-sequential programs, providing a bridge between requirements definition and implementation with the use of a design database that is automatically represented in Prolog. The current experimental version is implemented in C and Prolog, runs on a SUN workstation, and supports the iconic design of programs, their structural and temporal analysis, and automatic partial Ada source code generation.

CAEDE provides the basis for an SDE. It can be extended to support the programming and verification-testing phases with the addition of tools like syntax directed

Figure 3.1: The waterfall model of the software life cycle

Figure 3.2: The proposed use of the monitoring tool in the rapid prototyping process

editors, debuggers, and performance monitoring tools.

Figure 3.3 gives a high level view of the rapid prototyping environment. This figure's blocks have one-to-one mapping with the blocks in figure 3.2, after the specifications stage. Development in this environment starts after the requirements definition and specification, at the design level. The CAEDE design environment and methodology is used to iconically represent the system. In its current version CAEDE captures only the structural information of the design. An assumption can be made though, that the environment provides means to capture both temporal and functional behaviour of the system under design, and its data structures. It is then possible for the system's design to be automatically transformed into compilable units of HLL code. After the compilation, the system's executable images are downloaded to the target where execution is started. The Dynamic Performance Measurement system is connected to the target processor(s), collecting performance data which, after reduction and analysis, are fed back to the design workstation in graphical form.

This prototype DPM tool depends on certain services (CAEDE design and source code generation), and provides additional services (implementation quality assessment and execution monitoring). The value added is substantial since the previous level of service is greatly augmented.

For the software developer-user the environment is a compact system with a single interface (graphics, menu driven) and a consistent set of abstractions (tasks, rendezvous etc). For example, CAEDE uses parallelograms for the graphical representation of tasks on the screen and DPM uses bars in histograms to represent their execution times.

Figure 3.4 shows the building blocks of this idealized environment. In the first stage (the design of the system's structure), the user enters the system's objects and their interactions using menus of primitive icons while CAEDE is checking for consistency. The icons used at this stage and the methodology of their use are described in 4. The

Figure 3.3: The rapid prototyping environment with the proposed feedback connection

Figure 3.4: The idealized, CAEDE based SDE.

design is done in levels in a hierarchical manner. At each stage logical properties are automatically associated with the icons. These logical properties are used to transform the information on the screen into Prolog facts which are kept in a design database. Prolog tools can be used at this point to give the designer the first feedback on the quality of his design. Specifying and reasoning about temporal behaviour is done in terms of Prolog rules. For example, a temporal representation and analysis tool can be used to study the temporal aspects of the iconic design (detect the potential of task deadness). The same Prolog facts from the database can be used for the automated translation of designs into partial Ada programs. This is being possible due to the provision of the Ada language abstractions for concurrency and modularization. It is assumed that program fragments can be entered under the control of the iconic interface, and that there is a way of specifying external as well as internal temporal behaviour and functionality of components at each design level. In this way the code generation section of the environment is able to produce compilable code.

As a result, the costs for the implementation of the system are reduced drastically, and changes in the structure or the temporal behaviour can be done in the same interactive way, easily and cost effectively.

At the time when an implementation of the system is running on the target the designer can request information on its performance using the dynamic performance monitoring tool. DPM could take information from the design logical structure relating to the components of the design and their interactions, and combine it with information from the compiler's symbol tables on the position of the components in the target's memory. It then forms measurement setup files which are used by the logic analyzer to acquire data from the target processor(s) to which it is connected. The measured data are reduced partially by the logic analyzer and partially by the DPM to produce abstract information on the system. This information is graphically presented in a

histogram form on the designer's terminal. The percentage of time the processor spends executing a particular task is displayed relative to other tasks. The mean duration of specific interactions (rendezvous or interrupts service) can be displayed as well as the distributions. If those results show divergence from the system requirements then changes can be made to the structure or the functionality of the design which results in a new automatic implementation. This procedure is repeated until the system meets its requirements.

The environment also provides means for the system performance prediction. In Figure 3.4 together with the Prolog tools a modelling (*p-model*) tool is shown. P-model [7] can be used to predict a system's performance right after its structural definition. Thus, modelling the performance of a system under design is possible to a certain degree early in the design process. System behaviour reasoning tools, together with performance modelling, can be used during the early design stages in this environment providing the *pre-implementation feedback* on the design's quality and correctness.

Another use of the DPM tool becomes obvious now. The tool can be used for the verification of the estimations from the use of the p-model tool. Table 3.1 summarizes the differences and similarities between the two different performance tools of the environment.

Another role of these tools is to validate the developed system, that is, to show conformance to the requirements specification. They can be used to prove that the system meets its constraints, and its functional (hard) requirements. All these requirements are equally crucial in real-time embedded systems. An environment like the proposed one, fully automates the verification process in both design and implementation phases of the classical waterfall model.

The maintenance phase is also supported using the monitoring tool. Modifications to a delivered implementation could take place later in its life time, if for any reason has

	p-model	D P M
Inputs	Model of the system under development or simulation package Information about the workload and the temporal behaviour of the system (probably estimated or taken from the requirements specs)	The software system implemented, running on the target
Outputs	Performance data Utilization of S/W resources	Performance data Utilization of H/W and S/W Graphical representation of performance issues (histogramming etc)
Applicability of results	For a wide range of parameters or specified sets of parameters	For the special application only
Validity of results	Dependent on the model Accuracy need not have statistical dependence Accuracy dependent on the runtime of simulation	Dependent on the workload Very accurate if testing takes place when target is connected to the real-world
Advantages	Performance prediction over a wide range of parameters	Accurate performance measurement of a special system Does not alter the system under test Measures H/W as well as S/W characteristics
Disadvantages	Predicts, does not measure	Very costly to implement

Table 3.1: Comparison of the CAEDE's performance modelling tool with the monitoring tool.

been detected that it does not perform well. This detection can be done connecting the DPM tool to the system since this connection is external and passive.

The design environment has been used until now only in the design of Ada programs. However, neither CAEDE methodology [4], nor its notation is restricted to Ada. The design in similar rendezvous based concurrent languages is equally well supported, moreover, when those languages-systems are not as rich in features as Ada.

3.3 The Prototype Software Development Environment

Due to restrictions existing at the time this research was taking place, a significantly limited implementation of the above presented SDE was used.

The provided execution environment is running Harmony/C, whereas the CAEDE code generator produces only Ada code and that in skeleton form. Thus, the developed applications had to be implemented manually in Harmony/C, and only their structural design was done using the CAEDE graphics front-end.

This limited SDE uses the SUN workstation as a host for program development (manual), iconic system's structure design, and graphical performance results presentation. The host is connected to the execution environment and to the logic analyzer through its two serial ports.

A description of both software and hardware characteristics of the execution environment is given in the following subsection. The influence of these characteristics on the design of real-time software systems, and on performance monitoring of their execution is highlighted.

3.3.1 The Execution Environment

The Harmony Operating System

Harmony is a multitasking, multiprocessing operating system for real-time control [28]. It is mainly written in C and user programs must also be written in C. The latter though, do not look like usual C programs. Harmony consists of a kernel and a collection of tasks. It provides a message passing mechanism for intertask communication and synchronization. Although this mechanism is implemented as send-receive-reply construct, it is similar to Ada rendezvous. The Harmony kernel consists of a collection of primitives which are implemented in assembly language and C. Communication and synchronization primitives are implemented as functions. In Figure 3.5 the Harmony internals are shown together with a user program. The notation used in this figure is SDWA, where tasks are symbolized with parallelograms, boxes represent groups of tasks, and small boxes are Harmony OS kernel primitives. User multiprocessor programs are linked together with a copy of Harmony tasks and primitives to produce executable images ('dnd image' in fig. 3.5), which are then downloaded to the multiprocessor for execution.

The main features of Harmony are

- *Hierarchical parent-child task relationships.* The first user task, main, is created by Harmony. The Local-task-manager (Ltm), Directory (Dir), and Gossip tasks are also created by Harmony at start up. The main user task is thereafter responsible to create whatever tasks are needed calling `create` primitive. It also creates any device servers that are needed, calling `server-create` primitive.
- *Extensive device support via servers.* In general, user tasks do not use resources directly [27]. There are tasks that have given the responsibility for some resource, and are referred to as *servers*. Servers can be created to support exclusive handling of various devices (I/O, calendar clock, virtual terminals, etc.). In fig. 3.5 the I/O

Figure 3.5: The embedded system software.

server is shown (Sios Serv) which includes the input and output tasks (Tti and Tto respectively).

- *Fixed priority scheduling.* Each task has a fixed priority level. There are separate FIFO ready queues for each priority level on each processor. Every task which is not blocked, is in one of those queues. The first task on the highest priority nonempty queue is the active task, i.e. the one which is executing.
- A task continues to execute until it blocks or is preempted by a higher priority task becoming ready. That is, *Harmony tasks run 'natural break' and support preemption* [28, p.6].
- *Dynamic task creation.* Multiple instantiations of a specific task can be created dynamically (as in the case of servers).
- *Task migration among the processors is not supported* (static load balancing).

From a performance measurement tool point of view, these two issues of dynamic task creation and fixed task/processor bindings are important for several reasons. If tasks are dynamically created, it is difficult if not impossible to measure the data memory accesses of a task. Other methods must be used to estimate processor resource usage by a task. The lack of task migration simplifies the processor resource estimation for a collection of tasks, because it is known that they always execute on a certain processor.

The Multiprocessor Target Hardware

Multiple microprocessors have become a common element in current real-time systems design. The basic reasons for using multiprocessors in real-time systems are to guarantee responsiveness, to increase total computing capacity and to have redundancy to provide reliability. The multiprocessor system that is used here is a tightly coupled multiprocessor with linear shared memory. It consists of a VME chassis and a maximum

of eight processor cards. These DY-4 DVME-102 cards each have the MC68010 processor running at 8 MHz, ROM monitors, 256K of DRAM, serial I/O, timers, fully dual ported memory and I/O, and optional memory management. There exist some specific hardware features that support the Harmony Operating System. These are:

- The collection of processors is configured to have a linear address space. This allows completely shared memory space using a unique address for each location. The system's memory map is shown in Figure 3.6.
- VME bus arbitration and access is provided for each card.
- Interprocessor interrupts are provided using a write to a processor specific 'magic' location. This allows each processor to interrupt every other processor in the system including itself.
- By virtue of the use of the MC68010 processor, both test-and-set and multiple interrupt levels are provided.

The part of the software development environment presented in this chapter, which was chosen for implementation, is the monitoring of the real-time execution. The tool which was developed, attempts to close the loop from the target to the design workstation, providing the 'developer' with a real-time summary of the system's execution. The target environment as it is above described, together with some additional instrumentation, has been used for testing and demonstration of the dynamic performance measurement tool. The next ~~chapter~~ reports on the development of that tool, explains its use, and gives an application example.

Address in hex	
FF8800	Processor 7 I/O
FF8700	Processor 6 I/O
FF8600	Processor 5 I/O
FF8500	Processor 4 I/O
FF8400	Processor 3 I/O
FF8300	Processor 2 I/O
FF8200	Processor 1 I/O
FF8100	Processor 0 I/O
FF8000	
800000	Processor 7 (DRAM)
700000	Processor 6 (DRAM)
600000	Processor 5 (DRAM)
500000	Processor 4 (DRAM)
400000	Processor 3 (DRAM)
300000	Processor 2 (DRAM)
200000	Processor 1 (DRAM)
100000	Processor 0 (DRAM)
000000	

Figure 3.6: The mapping of the multiprocessor linear memory address space onto the various processors.

Chapter 4

Design of an Experimental Prototype

4.1 The Dynamic Performance Monitoring system

The Dynamic Performance Monitoring system takes advantage of some unique features of the Tektronix 1240 logic analyzer which constitutes its hardware instrumentation part. This instrument gives the user the ability to specify hardware or software events, and measure the activity levels of software modules (of a sequential program on a single processor), by means of a performance package. This is achieved mainly because of the fact that the logic analyzer is connected directly to the target processor chip via a probe, without interfering with its normal operation. Moreover, the performance analysis program that is stored in a ROM pack on the logic analyzer provides the functions of high level event detection and immediate data reduction.

Figure 4.1 shows the logic analyzer, its internals and its position in the environment. The use of DPM starts with the construction of the logic analyzer setup files on the host which are sent via an RS-232 line to the logic analyzer. These setup files are then used by the performance analysis program that is stored in the ROM pack to define the events to be measured. The logic analyzer then starts sampling the target processor lines to detect the events and collect the data. A first quick data reduction takes place inside

Figure 4.1 The logic analyzer and its position in the environment

the logic analyzer at the same time. This is required because the volume of the acquired data is huge, and there is no adequate local memory space to store them all, for later processing. This method saves a lot of work from the later analysis stages, but limits the kind of measurements that can be done to a small subset of the possible ones, since performance analysis programs are proprietary software in ROM.

DPM Version 1.0, 1.1 and 2.1 extend and enhance the use of the logic analyzer to be able to measure task activity on a multi-tasking system, by considering the task executable code as a procedure in a sequential program. They use the concept of taking the compiler or assembler symbol table, to correlate mnemonics with target memory addresses. In Version 2.1 another method is being used to estimate processor resource usage by a task, which provides for direct measurement of the time that the task is active. The following subsections report on both these methods.

Figure 4.2 shows how hardware is organized in the ARTT laboratory. Both ver-

Figure 4.2: The ARTT laboratory hardware

sions 1.0 and 1.1 of DPM used only the DSM6816, Logic Analyzer, Target system, and graphics terminal independently of the overall SDE. Versions 2.0 and 2.1 use the whole system as it is shown in this figure, with the main load on the SUN workstation. This specific hardware layout is completely experimental and subject to changes. For a final implementation of the proposed rapid prototyping environment, the SUN, Target, Logic Analyzer, and In Circuit Emulator parts are the only ones required.

4.1.1 DPM Version 1.0

The first version of the tool was limited in scope. The objectives of this version were

- The construction of a prototype that will serve as a feasibility study
- Provision of a demonstration system and an experimental testing facility
- Development of familiarity with hardware instrumentation
 - Exploration the capabilities of the logic analyzer
 - Proof of the effectiveness of remote control of the instrument
- Determination of a proper user interface

The design and construction of the system began with incomplete specifications, therefore the rapid prototyping methodology applied to the development of DPM. The tool was built modularly and in an incremental fashion. Modules were added to the prototype as they were developed. This approach allowed preliminary implementations of the prototype to be up and running quickly, long before the final version was available. The development of this version had been done on the DY-4 DSM 6816 system running UniPlus[®] UNIX System III¹. At that time it was the only development system in the ARTT laboratory. This tool implementation had two main functions. First it

¹UniPlus[®] is a Trademark of UniSoft Corporation of Berkeley.

automatically extracted information about the application tasks from the symbol tables, and constructed the setup files that were downloaded to the logic analyzer. This function integrates the logic analyzer with the tool, and saves the user from the need to program it through the front panel control keys, a complicated and error prone procedure. After the execution, it recovers the data from the logic analyzer output files, and uses this data to plot the histogram graphs. The fundamental input to its graphics subsystem is the Logic Analyzer data files resulting from the measurements. The format of the output on the graphics terminal is a histogram representation of the processor resource usage by task for the different processors. For this representation the use of 4010 termcap with tplot and graph utilities was used. The tool stores the data for the graphics output in files on the host. In this way the user is able to recall and study older measurements, or even to compare them with new measurements by having the ability to display them on the screen simultaneously.

The tool was provided with a menu driven user interface which unifies the two basic parts of the tool, the data acquisition and logic analyzer setup on one hand, with the graphics generation on the other. It is designed to be modular, so it is easily extensible, and it is possible to switch parts, without discarding the whole system.

Quick implementation was facilitated by the UNIX tools, which helped to handle successive versions of the prototype. UNIX uniform treatment of files, and powerful command language cshell helped in the easy connection of modules as they developed.

The tool calculates the processor resource usage by the various tasks, considering tasks as pieces of executable code in the target's memory, and measuring the percentages of instruction fetches from the corresponding parts of memory. This imposes a difficulty when dealing with the idle task which is 'running' when no other task is running on the processor. Measurements are taken sampling the processor lines, and are biased when there is an idle task 'running' on the processor, which executes a STOP instruction (that

is, it does no fetches).

4.1.2 DPM Version 1.1

The biasing in the measurement results was a problem which has found solution in the second version of the tool (Version 1.1). The current Harmony implementation uses a STOP instruction in the idle loop to reduce memory contention; however, this makes idle time measurement more complex since the resource estimator uses memory instructions accesses, and the processor is frozen in the Idle task. Unless further steps are taken, a distorted picture of the processor resource usage is presented.

The method used to improve the estimates uses a count of occurrences of memory instruction fetches for each task in the system. The idle task is included in this count. Now, the idle task occurrence value is increased using a multiplier. This multiplier is calculated using the mean time the idle task was idle, effectively adding the correct number of occurrences to the idle process count. The sum of all counts gives total execution time and the percentage values for tasks can be directly calculated. The mean idle time of the processor is directly measurable as the percentage of time the idle process executes (the time between the fetch of the STOP instruction and the next fetch).

A detailed description of version 1.1 of the prototype system can be found in [67], where its architecture and operation is explained and an early application of its use is reported.

The prototype DPM versions 1.0 and 1.1 did not meet all the requirements for a performance analysis tool. Implemented only the most important aspects of the measurement part of the tool, providing some feedback from the target on system's performance. After all, a prototype system should be considered to be a throw away system, and the final system built from scratch. [26]

4.1.3 DPM Version 2.0

The DPM Version 2.0 resulted from the porting of the tool software to a SUN workstation and the addition of the timing measurements feature. As a result of the port to the high quality graphics workstation, dramatic improvements in the user interface (SUN-core graphics, menu item selection using mouse), and more portability were achieved. Time measurements give the user the ability to obtain mean values of critical timing considerations as well as their distribution graph. The distribution of the response times to a specific interrupt (latency), as well as messaging delay times are now displayed on the screen together with the task runtime percentages histogram.

Another measurement issue is the critical estimate of VME bus utilization. Often, bus saturation is a major limitation for multiprocessing target systems with a single shared bus. The measurement or estimation of bus saturation is a critical parameter. The tool measures VME bus utilization by observing the VME arbiter operation on processor 0. It then displays a distribution graph of duration of the VME bus accesses. A detailed description of version 2.0 can be found in [70]. The 'dinning philosophers' problem is presented there as a multiprocessor Harmony application, to demonstrate the workings of the tool. DPM is able to measure task activity on several processors on such multiprocessor applications by switching the logic analyzer among the target processors and repeating the application's execution. There is a hardware imposed difficulty in the way the tool is presently implemented. There is no smooth way to switch the logic analyzer among the processors. If the logic analyzer probe could be connected to a box which has a switch and several dummy probes which can be connected to every processor, then the target software would need to be downloaded only once. After each execution, the switch can be moved to the next processor.

There is a disadvantage in all the above versions. The measurements of task activity do not cover neither the case of global procedures and system primitives, nor the case of

a system that uses DMA. These disadvantages do not exist in version 2.1, since a new measurement method has been devised.

4.1.4 DPM Version 2.1

In multitasking systems the highest priority task is given the CPU, making the *state transition* from the ready state to the running state (*context switch*). This *dispatching* in *event driven* multitasking systems, like the real-time systems, is done depending on interrupts. Every system has its own way of dispatching processes depending on its internal philosophy. State transitions it is possible to be detected externally in all cases. In Harmony for example, the *task descriptor* address of the task which is in the running state each time, is written at a special memory location (*Active*), at activation time.

The new method for the measurement of the processor time consumption by the various tasks is based on this fact. First, DPM uses the logic analyzer to 'read' the task descriptors pointers of the running tasks on a processor. This is done by specifying the event of WRITE on the special location *Active* and storing the data at this moment. The value of *Active* is known to the DPM, (taken searching the compiler's symbol table), and it is standard for all Harmony programs, since Harmony code is always linked first. DPM associates then the task descriptor pointer with the task names, using the sequence of creation, and the fact that these pointers point to task descriptors sequentially arranged on the stack. It then uses this data to specify the event of a WRITE of a specific task descriptor value to the *Active* location, and the event of the next WRITE of any other task descriptor to that location. DPM starts the logic analyzer to measure the mean time between those two events (t_i). The t_i values are the mean active times for each task (the time between activation and the next context switch, or the next WRITE to *Active*). Then it measures the number of occurrences of the WRITE of a specific task descriptor pointer to the *Active* location over a certain number of any WRITES (n_i) to that location. The mean task active time of all the tasks is measured at the same time.

(t_m), as the time between any two successive task switches. DPM then computes the percentages of the active time for each task ($t_{i,a}$) from :

$$t_{i,a} = \frac{t_i * n_i}{t_m}$$

and plots the histogram of those values on the screen. It would be easier to compute these percentages, if the logic analyzer was giving the ability to measure occurrences of events over a certain period of time. Having then the frequencies of the task activations, the task active times could be computed multiplying by n_i .

The above method requires more complicated interaction between the DPM software and the logic analyzer, but produces more accurate results. It measures the idle processor time directly, by measuring the Idle task 'run time'. It takes into consideration the Harmony primitives, 'charging' their run time to the task which is calling them. The same reasoning charges the run time of any global procedures in the program to the task that calls them. In this way global procedures and Harmony primitives introduce no more distortion in the task activity measurements (neither does any DMA).

Another reason this method is more powerful, is that it covers efficiently the case of new systems based on processors which provide for hardware context switching (like MC68020 and iAPX 286). In this case what is needed is the monitoring of the execution of the special 'context switch instructions' to directly identify tasks becoming active, and measure their active time.

4.2 Testing

A small application was chosen and developed in the prototype environment for a demonstration of its functions. It is the classical bounded buffer example for concurrent systems. The system consists of three user tasks. One of them, the Buffer, has exclusive access to a memory area, and the others are the producer (main), and the consumer. The producer produces characters (in this case), which puts in the buffer one by one.

Figure 4.3 Timing diagram for the Harmony C example

The consumer takes and consumes characters (in this case it collects and displays them as strings on the screen). The role of the Buffer task is to smooth any variations between the speed of output of the producing task, and the speed of input of the consuming task, providing mutual exclusion on its data structures. In many systems supporting concurrency there are other ways of ensuring mutual exclusion with the use of semaphores, monitors, mailboxes, etc. In the Harmony C case, a task is used. The same holds for the Ada case. The advantage of this approach is that all task intercommunication and synchronization activities are performed by active program objects at the same logical level. There is no hidden functionality buried in signals or messages [4, p. 22].

A timing diagram showing the interactions between the above three tasks is shown in Figure 4.3. The design of this system's structure is shown in Figure 4.4 as it is drawn on the SUN workstation screen using the CAEDE system's graphics front-end. The three system tasks are shown as black boxes (parallelograms). Other information included in

Figure 4.4: The producer - consumer system design.

this icon is the interactions between the tasks, (arrows from the sending task pointing to the receiving one), and the data flow (characters 'moving' on these arrows). The design was coded manually in Harmony/C, due to the fact that the environment does not produce compilable code. A first quick implementation is included in Figure B.1 in Appendix B.

The same example implemented in Ada can also be found in the Appendix B, Figure B.2. In the Ada case the use of the **selective wait** and **guard** features make the implementation far more efficient, (based on the assumption that both C and Ada compilers produce code comparable in efficiency). The significant difference, as regards to the system performance, is that in the case in which the producer gets too far ahead of the consumer, waits in the PUT entry queue of the Buffer until the consumer calls the GET entry. This results in a change to the guard, and opens the PUT entry. In this Harmony implementation though, in the same situation, producer (main) and Buffer, rendezvous continuously until the Consumer calls Buffer (the producer is *polling*). This introduces unwanted overhead which slows the system significantly. However, changes can be done for an improved implementation. The system can use a flag to signal 'buffer full', and in this way the buffer task will know when to *send-block* the producer, preventing him from polling. This change will result in less time consumed by the producer task, difference

which will show up in DPM's histograms.

The performance measurement results are shown in Figure 4.5 as they produced by the DPM for the herein presented application. For the histograms and the distribution graphs a special graphics library and filters were used. The parameters that are measured and displayed on the screen are the CPU time consumption, and the rendezvous times. The histogram on the top of Figure 4.5 shows the percentage of the processor time consumed by some tasks in the application system. The vertical axis represents the various tasks whereas the horizontal axis represents the time spent by these tasks. The top section of the histogram shows the Processor 1 time spent by the Consumer (cons.1) and Idle (idle.1) tasks. The lower section of this histogram shows the Processor 0 time spent by the Buffer (buf.0), main, Sios Server (iosr.0), and idle tasks. The distribution graph in the lower part of Figure 4.5 shows the Consumer rendezvous duration. It is illustrated that there are measured 150 rendezvous of 1 msec duration, 380 of 1 sec, 100 of 2 secs, etc. over a certain measurement period.

Changes in the implementation can lead to different performance results. For example, if the delay loop which exists in the Consumer task changes, (see Figure B.1) making the Consumer working at about the same speed with the Producer, there is more synchronization in the system. This will result in less processor time spent by the Producer task, which now needs less attempts to put a character in the Buffer. This change in the consumed CPU time and the number of rendezvous will be displayed on the screen by the DPM tool.

For the prototype environment implementation the SUNtools were used to divide the screen into three working areas (windows). Figure 4.6 is a screen dump from the SUN workstation. The larger window is for the use of the CAEDE graphics system, the top one is used for editing the manually coded programs, and the right one is used for calling the DPM tool to conduct the measurements and display the performance information.

Figure 4.5: The graphics output of the DPM tool.

- 12 Cheatman, Jr. T.E. 'An overview of the Harvard Program Development System' in *Software Engineering Environments*. ed. H. Hünke, North-Holland Pub Co, 1981 (Proc. Symposium in Lahnstein, Germany June 1980), p.253-279
- 13 Cheatman, Jr. T.E. 'Comparing Programming Support Environments.' in *Software Engineering Environments*. ed. H. Hünke, North-Holland Pub. Co, 1981 (Proc. Symposium in Lahnstein, Germany June 1980), p.11-25
- 14 Cheriton, D.R. *The Thoth System: Multi-Process Structuring and Portability*. North-Holland, 1982
- 15 Cho, C.K.; Lin, E.K.; Jen, C.L. 'On Performance Evaluation of Multiprocessor systems for RT Simulation' *Proc. of the 17th Annual Simulation Symposium*, IEEE, Tampa FL, March 1984, p.209-225.
- 16 Conner, M.H.; Parr, F.N. 'Portable, secure Communications Software' *IBM Research Report RC 8875 (#98642)*, IBM, N.Y., Apr 1981
A rough description of the NIL language. For information on how S.W. development is supported by NIL see p. 73
- 17 Cristofor, E. 'Automatic generation of the test Software' *SOFTAIR*, IEEE CS Press 1983, p.129-44
- 18 Delisle, N.M. et al. 'Viewing a programming environment as a single tool' *SIGPLAN Notices*, Vol 19(5), (May 1984) p.49-56
- 19 Deutch, P.L. and Taft, E.A. 'Requirements for an experimental Programming Environment' *Xerox PARC Report CSL-80-10*, June 1980
- 20 Deutch, M.S. 'An industrial software engineering methodology supported by an automated environment' *Proc. NCC, 1982*, p.302-307
- 21 Elwell, J.F. 'An approach to the definition and implementation of a software development environment.' *Proc. NCC, 1982*, p.309-318.
- 22 Ferrari, D. *Computer Systems Performance Evaluation*, Prentice-Hall, Englewood Cliffs, NJ, 1978
A comprehensive treatment of all the issues related to performance evaluation. It explains measurement, simulation, and analytical techniques.
- 23 Fidge, C.J.; Cain, C.J.; Jackson, L.N.; Pascoe, R.S.V. : 'An overview of the MELBA automatic code generation project.' *A.T.R. Vol.18, No.1*, p.3-12. Melbourne Australia 1984.
- 24 Fischer, G.; Boecker, H.-D. : 'The nature of design processes and how computer systems can support them.' *Integrated Interactive Computing Environments, ECICS* eds. Degano, P.; Sandewall, E., North-Holland, 1983
- 25 Fritzson, P. : 'Preliminary experience from the DICE system a distributed incremental Compiling Environment.' *SIGPLAN Notices*, Vol 19(5), May 1984, p.113-123

Chapter 5

Conclusions and Recommendations

This chapter concludes the thesis, summarizing its contribution, and indicating further work which will lead to the implementation of *the integrated feedback-directed SDE*.

5.1 Conclusions

The research reported in this thesis advances the discipline of Software Engineering at both conceptual and practical levels.

Primarily, it reports on the state-of-the-art in the area of software engineering environments, identifying the widespread conception of programming environment, and explains how this conception has been challenged by the new SDEs. The need for integration and feedback during development is identified as well, and the existing methodologies and methods used to measure, monitor, and evaluate systems performance are presented.

The use of a monitoring tool in the rapid prototyping approach was proposed. It resulted from the need for extended support during every development phase. An idealized environment, which was described supports and unifies all development phases, centered around a SUN workstation, accessed via a single consistent graphics, menu-driven

interface. It is especially well suited for the development of complicated concurrent real-time systems where performance can be considered at least equally important with the functionality of the system. With this method, reduction of both time and complexity required to develop software products is achieved.

Assuming the existence of the other parts of the environment, a tool was built, which gives the designer the ability to see the implications on performance of alternative design and implementation considerations. The DPM system adds substantial value to the existing environment since it unifies design and testing, providing a bridge between the host and the target in the case of embedded systems. Its main advantage is that the developer, being at the design stage, can easily modify the design, which will be again automatically implemented, and see in this way the results of his modifications in terms of performance measures. DPM conforms with the need for extended support during the software development by automating as many performance measurements as possible, providing an integrating function in the framework of the development environment.

5.2 Future Work

The success of the limited application of the principles of the thesis suggests that it would be beneficial to pursue them further. The environment presented in the two preceding chapters can be improved in several ways as regards to the enhancement of the monitoring tool. A summary of these improvements follows.

- More integration of the DPM system with the development environment is needed.

This can take place in several ways.

The connection of the DPM user interface with the CAEDE user interface. They are both mouse - menu driven using SUNcore software.

The user should be able to define events pointing with the mouse on the structure graph. For example, pointing to the S1 (send) point on the design (see Figure 4.4),

and asking for the mean time and distribution of the in-rendezvous delay time for the Consumer task. This involves in having introduced the S1 symbol in the implementation code and by pointing to it, initiating a measurement from the fetch from this location in the task's code until the fetch of the next after this send instruction.

- The logical structure manager information has to be able to be passed to the DPM tool. DPM will keep a main data structure which will be directly related to the structural information on the CAEDE picture. It can be used in conjunction with the logical structure manager and store information on every object in the picture, namely the type, the id, and the performance results. The latter will be available only after the designer has asked from the DPM to compute them, and will be graphically displayed on the screen. Multiple performance results will be easily stored and recalled for display for comparison purposes.
- The tool itself has to become multitasking. The main parts of the DPM tool, namely the setup files construction, the communication with the logic analyzer, and the presentation of results, it is possible to be combined in time. This will make possible the combination of space-time measurements in interleaved time periods. A continuous update of the histograms can also then be possible.
- The obstacles resulting from the uncomplete documentation on the logic analyzer workings, both hardware and performance software, have to be removed. In general, the specific logic analyzer used, has several limitations with regard to the kind of measurements it can be programmed to perform. An instrument with more advanced features is needed. It must be able to detect and measure parallel events. The availability of a higher capacity connection between the instrument and the environment's host, would be a very useful feature drastically reducing the setup

and data file transfers

- Finally, small modifications to the software of the specific target environment have been examined. Those belong to the case of software instrumentation. It seems possible that a few small modifications to the Harmony Operating System would result in the DPM getting more information on task behaviour on the processor. This involves modifying Harmony primitives to keep accounting information in special memory locations, accessible by the DPM tool.

All the experimental work reported in this thesis was conducted in a specific computing environment. The reason was that the proposed approach was based on and required the existence of certain features, however, this does not mean that it can only be applied to this specific environment. Any one of the state-of-the-art high quality, workstation based, UNIX-like environments could be the host for an implementation of the proposed techniques. Moreover other environments features could proved to be helpful for enhancements in the proposed prototype. Other targets' features and richer hardware instrumentation would also be valuable. Thus, it would be worth-while to examine how the proposed approach could be followed elsewhere.

Bibliography

- 1 Arthur, J.D. ; Corner, D.E. : Omni : An interactive programming environment based on tool composition *Proc 8th COMPSAC'84, IEEE CS Chicago, Ill Nov 1984* p 28-36
- 2 Arthur, J.D. ; Reed, D.A. : Prometheus : An interactive environment for the development and parallel execution of functional programs *Proc 8th COMPSAC'84 IEEE CS Chicago, Ill Nov 1984* p 44-53
- 3 Boari, M. et al : Multiple-Microprocessor programming techniques : MML, a new set of tools *IEEE Jan 1984* p 47-58
- 4 Buhr, R.J.A. : *System Design With Ada* : Prentice-Hall : N.J 1984
- 5 Buhr, R.J.A. : Selected systems Design with Ada Projects, Fall 1984 *Report SCE-85-4, Dept of Sys and Comp Eng, Carleton Univ, 1985*
- 6 Buhr, R.J.A.; Karam, G.M.; Woodside, C.M. : An overview and example of application of CAEDE : A new experimental design environment for ADA in *Ada in use, Proc Intl Ada Conf, Paris, France, May 1985*, p 173-184
- 7 Buhr, R.J.A.; Hayes, C.J.; Karam, G.M.; Neron, E.; Van der Loo, K. : CAEDE User's Guide, Version 1.2 *Report SCE-85-12, Dept of Sys and Comp Eng, Carleton U, Ottawa, May 23, 1985*
- 8 Burford, M.A.J.; Belli, F. : CADAS : A tool for rapid prototyping and testing of embedded software. *Symp. on Applications and Assessment of Automated tools for software development, IEEE 1989*, p.27-33
- 9 Buxton, J.N. and Druffel, L.E. : Requirements for an ADA Programming Support Environment : Rationale for STONEMAN. *COMPSAC'80, Chicago IL, Oct 1980*, also in *Software Engineering Environments*, ed. H. Hünke, North-Holland Pub. Co., 1981. (Proc. Symposium in Lahnstein, Germany June 1980), p.319-330
- [10] A special issue on Smalltalk. *BYTE, Vol.6(8)*, (Aug 1981).
- [11] Cheatman, T.E. et. al. : 'A system for program refinement.' *Proc. of the 4th Intl. Conf. on Software Eng.*, San Diego (1981), p.430-437.

- 12 Cheatman, Jr. T.E. 'An overview of the Harvard Program Development System' in *Software Engineering Environments*. ed. H. Hünke, North-Holland Pub Co, 1981 (Proc Symposium in Lahnstein, Germany June 1980), p.253-279
- 13 Cheatman, Jr. T.E. 'Comparing Programming Support Environments.' in *Software Engineering Environments* ed. H Hünke, North-Holland Pub Co, 1981 (Proc Symposium in Lahnstein, Germany June 1980), p.11-25
- 14 Cheriton, D.R. *The Thoth System Multi-Process Structuring and Portability*, North-Holland, 1982
- 15 Cho, C.K.; Lin, E.K.; Jen, C.L. 'On Performance Evaluation of Multiprocessor systems for RT Simulation' *Proc of the 17th Annual Simulation Symposium*, IEEE, Tampa FL, March 1984, p.209-225.
- 16 Conner, M.H.; Parr, F.N. 'Portable, secure Communications Software' *IBM Research Report RC 8875 (#98642)*, IBM, N Y, Apr 1981
A rough description of the NIL language. For information on how S.W development is supported by NIL see 73
- 17 Cristofor, E. 'Automatic generation of the test Software' *SOFTAIR*, IEEE CS Press 1983, p.129-44
- 18 Delisle, N.M. et al 'Viewing a programming environment as a single tool' *SIGPLAN Notices*, Vol 19(5), (May 1984) p.49-56
- 19 Deutch, P.L. and Taft, E.A. 'Requirements for an experimental Programming Environment' *Xerox PARC Report CSL-80-10*, June 1980
- 20 Deutch, M.S. 'An industrial software engineering methodology supported by an automated environment.' *Proc. NCC, 1982*, p.302-307
- 21 Elwell, J.F. 'An approach to the definition and implementation of a software development environment.' *Proc. NCC, 1982*, p.309-318.
- 22 Ferrari, D. *Computer Systems Performance Evaluation*, Prentice-Hall, Englewood Cliffs, NJ, 1978
A comprehensive treatment of all the issues related to performance evaluation. It explains measurement, simulation, and analytical techniques.
- 23 Fidge, C.J.; Cain, C.J.; Jackson, L.N.; Pascoe, R.S.V. 'An overview of the MELBA automatic code generation project.' *A.T.R. Vol.18, No.1*, p.3-12. Melbourne Australia 1984.
- 24 Fischer, G.; Boecker, H.-D. : 'The nature of design processes and how computer systems can support them.' *Integrated Interactive Computing Environments, ECICS* eds. Degano, P.; Sandewall, E., North-Holland, 1983
- 25 Fritzson, P. : 'Preliminary experience from the DICE system a distributed incremental Compiling Environment.' *SIGPLAN Notices*, Vol 19(5), May 1984, p.113-123

Describes the DICE prototype system, its features and performance, and gives the project origins and current status. Strategies for implementing portable SEEs are discussed and exemplified by the DICE system.

- 26] **Gehani, N.H.** : 'A study in prototyping.' *ACM SIGSOFT SE Notes*, Vol.7(5), Dec.1982, p.71-74
- 27] **Gentleman, W.M.** : 'Message passing between sequential processes the reply primitive and the administrator concept.' *Software Pract. Exper* Vol 11, 1981, p.435-466.
- 28] **Gentleman, W.M.** : 'Using the Harmony Operating System.' *National Research Council Canada NRCC No 23030*, Dec. 1983, revised Oct 1984.
- 29] **Goldberg, A.** : 'Smalltalk-80. The Interactive Programming Environment'. Addison-Wesley, Reading, MA., 1984
- 30] **Gomaa, H.** : 'A software design method for RT systems-DARTS' *CACM*, Vol.27(9), (Sept 1984), p.938-949
- 31] **Gotlieb, C.C.; MacEwen, G.H.** : 'System evaluation tools' in *Software engineering techniques*, ed. Buxton-Randell, Report on a NATO conf., Rome, Italy, Oct 1969, p.93-99
- 32] **Haberman, A.N.** : 'An overview of the Gandalf project.' *Comp Science Research Review 1978-1979*, Carnegie-Mellon Univ, Pittsburg 1980.
- 33] **Haberman, A.N. and Perry, D.E.** : 'System Composition and Version Control for Ada (SVCE)' in *Software Engineering Environments*. ed. H. Hünke, North-Holland Pub. Co., 1981. (Proc. Symposium in Lahnstein, Germany June 1980) , p.331-343
- 34] **Hackler, W.R.** : 'USE.IT' for real-time control' *AIAA Computers in aerospace IV Conference*. Hartford, Connecticut, Oct.1983, p.72-79.
- 35] **Hansen, H.-L. and Müllerburg, M.** : 'Software Engineering Environments. A bibliography.' in *Software Engineering Environments*. ed. H. Hünke, North-Holland Pub. Co., 1981. (Proc. Symposium in Lahnstein, Germany June 1980) , p.359-410
- 36] **Hansen, P.B.** : 'A keynote address on concurrent programming.' *COMPSAC '78*, Chicago, IU., Nov. 1978.
- 37] **Hayes, C.** : 'CAEDE User's Guide.' *Report SCE-84-24, Dept of Sys. and Comp. Eng.*, Carleton Univ., Nov 1984
- 38] **Heildenberger, P.; Lavenberg, S.S.** : 'Computer Performance Evaluation Methodology' *IEEE Trans. on Computers*, Vol c-33(12), Dec 1984, p.1195-1220.
- 39] **Henderson, P.** : 'Finite State Modelling in Program Development.' *SIGPLAN Notices*, Vol.10(6), (June 1975), p.221-227

A report on the experimental TOPD development environment, used in a software eng. course at the Univ. of Newcastle upon Tyne.

- 40 Howden, W.E. 'Contemporary software development environments' *CACM* Vol.25(5), p 318-329, (May 1982)
- 41 Jackson, M.A. 'System Development', Prentice-Hall Intl., Englewood Cliffs, 1983
- 42 Kandt, K. 'Pegasus: A tool for the acquisition and reuse of software designs' *Proc 8th COMPSAC'84, IEEE CS Chicago, Ill* Nov 1984 p 288-293
- 43 Kernighan, B.W.; Pike, R. *The UNIX Programming Environment* Prentice-Hall, Inc., N J 1984
- 44 Kobayashi, H. *Modelling and Analysis: An introduction to system Performance Evaluation Methodology* Addison-Wesley, Reading, MA
A formal, mathematical treatment of stochastic, simulation, and statistical modelling methodologies with applications to computer system performance
- 45 Lauer, P.E. and Shields, M.W. 'COSY: An environment for development and analysis of concurrent and distributed systems' in: *Software Engineering Environments*, ed. H. Hünke, North-Holland Pub. Co., 1981 (Proc. Symposium in Lahnstein, Germany June 1980), p 119-156
A tutorial on computer systems design, analysis and their implementation environment
- 46 Leblang, D.B. 'Abstract Syntax Based Programming Environment' *ACM Ada, TEC, Conf on Ada*, Wash. D.C., Oct 1982
- 47 Leblang, D.B. and Chase, Jr. R.P. 'Computer-Aided Software Engineering in a Distributed Workstation Environment' *SIGPLAN Notices*, Vol 19(5), May 1984, p.104-112.
A detailed description of the Domain SEE, a comparison with other systems and a discussion of issues involved in the construction of a highly reliable, safe, efficient, and distributed SEE
- 48 Liu, J.W.S.; Liu, C.L.; Miyazaki, N.; Yamazaki, H. 'Performance Evaluation of Multiprocessor Systems containing Special Purpose Processors' *Podstawy Stereomania*, Tom 10(3), 1980, p.201-222.
- 49 Marsan, M.A.; Balbo, G.; Conte, G.; Gregoretti, F. 'Modelling Bus Contention and Memory Interference in a Multiprocessor System' *IEEE Trans. on Comp.* Vol. c-32, No 1, Jan. 1983, p.60-72.
- 50 Matsumoto, Y. et al.: 'SWB System: A software factory.' in: *Software Engineering Environments*, ed. H. Hünke, North-Holland Pub. Co., 1981. (Proc. Symposium in Lahnstein, Germany June 1980), p.305-317
- 51 Medina-Mora, R. and Feiler, P.H.: 'An incremental Programming Environment.' *IEEE Trans. Software Eng.* SE-7(5), (Sept 1981), p.472-482.
- 52 Metzger, J.J.; Dniestrowski, A.: 'PLATINE- A S/W Engineering Environment.' *SOFTAIR, IEEE CS Press* 1983, p.129-44

- 53 **Mitze, R.W.** The UNIX System as a software engineering environment. In *Software Engineering Environments* ed H Hunke North-Holland Pub Co 1981 (Proc Symposium in Lahnstein Germany June 1980) p 345-357
- 54 **Montanari, U.** Towards an integration between language and software development environment. *Theory and Practice of Software technology* eds Ferrari D et al North-Holland 1983 p 195-202
- 55 **Moriconi, M. and Hare, D.F.** PegaSys: A system for graphical explanation of program designs. To appear in *ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments* June 1985
- 56 **Moriconi, M. and Hare, D.F.** Visualizing Program Designs Through PegaSys. To appear in *IEEE Computer*, Aug 1985
- 57 **Murakami, N. et al** SDEM and SDSS: overall approach to improvement of the software development environment. In *Software Engineering Environments* ed H Hunke North-Holland Pub Co 1981 (Proc Symposium in Lahnstein, Germany June 1980) p 281-294
- 58 **Nakamura, Y ; Miyahara, R ; Takenchi, H.** Complementary approach to the effective software development environment. *Proc COMPSAC '78 IEEE CS* Chicago Ill, Nov 1978, p 235-240
- 59 **Nassi, I. ; Shneiderman B.** Flowchart techniques for structured programming. *SIGPLAN Notices*, Vol 8(8), p 12-26, Aug 1973
- 60 **Pearson, D.J.** The use and abuse of a software engineering system. *Proc NCC*, N.Y., N.Y. June 1979, P 1029-1035
- 61 **Pierce, R.H.** Ada in the Eclipse project support environment. In *Ada in use*, Proc Intl Ada Conf Paris France, May 1985, p 309-320
- 62 **Plattner, B.** Real-Time Execution Monitoring. *IEEE Trans on SE* Vol SE-10(6), Nov 1984, p.756-764
A definition of the basic requirements for a real-time monitoring facility and a study of dynamic monitoring issues. One of the very few publications on the topic
- 63 **Ramamoorthy, C.V. et al.** 'Software engineering: Problems and Perspectives.' *IEEE Computer*, Oct.1984, p.191-209.
- 64 *Reference Manual for the Ada[®] Programming Language.* ANSI/MIL-STD-1815A-1983, U.S. DoD, Springer-Verlag, N.Y., Feb. 1983.
- 65 **Reis, S.P.** : 'Graphical program development with PECAN program development system.' *SIGPLAN Notices*, Vol 19, No 5, May 1984.
- 66 **Reis, S.P.** : 'PECAN: Program Development Systems that support multiple views.' *Proc. 7th Intl. Conf. on SE, Orlando, FL.*, March 1984 (IEEE) p.324-333
- 67 **Rowe, P.K.; Sartzetakis, S.S.; Vishnubhatla, B. :** 'A Multiprocessor Performance Measurement Tool.', *Proc. of 10th USENIX*, Portland, Oregon, June 1985.

- 68 Riddle, W.E. 'An assessment of DREAM in Software Engineering Environments' ed H Hunke North-Holland Pub Co 1981 (Proc Symposium in Lahnstein, Germany June 1980) p 191-205
- 69 Saltzer, J.H.; Gintell, J.W. 'The instrumentation of Multics' *Comm of the ACM* Aug 1970 Vol 13(8)
- 70 Sartzetakis, S.; Rowe, P.K.; Dodds, D.; Nightingale, J.; Saikaley, A.; Vishnubhatla, B. 'A Real-Time Multiprocessor Performance Monitoring tool to be presented in the Intnl Electrical and Electronics Conf ELECTRONICOM '85 IEEE Toronto Oct 1985
- 71 Snowdon, R.A. 'CADEs and software system development' in Software Engineering Environments ed H Hunke North-Holland Pub Co 1981 (Proc Symposium in Lahnstein, Germany June 1980) p 81-95
- 72 Standish, T.A. 'ARCTURUS: An advanced Highly-Integrated Programming Environment' in Software Engineering Environments ed H Hunke North-Holland Pub Co 1981 (Proc Symposium in Lahnstein, Germany June 1980) p 49-60
- 73 Strom, R. and Halim, N. 'A new programming methodology for long-lived software systems' *IBM J Res Develop* Vol 28(1), Jan 1984
- 74 Stucki, L.G.; Walker, H.D. 'Concepts and prototypes of ARGUS' in Software Engineering Environments ed H Hunke North-Holland Pub Co 1981 (Proc Symposium in Lahnstein, Germany June 1980) p 61-79
- 75 Stucki, L.G. 'What about CAD/CAM for software?' The ARGUS concept *SOFTAIR*, IEEE CS Press 1983, p 129-44
- 76 Svobodova, L. 'Computer Performance Measurement and Evaluation Methods Analysis and Applications', N.Y., Elsevier, 1976
A general introduction to measurement tools and methods of performance analysis and prediction
- 77 Taylor, R.N. and Standish, T.A. 'Steps to an Advanced Ada Programming Environment' *Proc 7th Intl. Conf on SE, Orlando FL, March 1984 (IEEE)*, p.116-125
Covers some of the features of the current working prototype of Arcturus SDE: Ada design, performance measurement and analysis.
- 78 Teitelman, W. and Masinter, L. 'The Interlisp Programming Environment.' *IEEE Computer* Vol.14(4), (April 1981), p. 25-33.
- 79 Teitelman, W. : 'A tour through Cedar.' *Proc. 7th Intl. Conf of Software Eng., Orlando, FL., (March 1984)* p.181-195.
- 80 Tichy, W.F. : 'A data model for Programming Support Environments and its application.' in: *Automated Tools for Information System Design and Development*, ed. : H.-J. Schneider and A. Wasserman, North-Holland Publishing Company, Amsterdam 1982

- 81 Tichy, W. F. 'RCS A Revision Control System' in *Integrated Interactive Computing Systems* ed P. Degano and E. Sandewall, North-Holland, P.C., 1983
- 82 Tsuji, J. et al. 'Dialogue management in the Personal Sequential Inference machine (PSI)' *Proc ACM'84 Annual Conf - The Fifth Generation Challenge*, ACM Oct 1984
- 83 Tucker, M. 'UNIX emerges as a Universal Tool Kit' *Mini-Micro Systems*, Vol 18(2), (Feb 1985), p 149-160.
- 84 Vrsalovic, D.; Siewiorek, D.P. 'Performance analysis of multiprocessor based control systems' *Proc. of the Real Time Systems Symposium*, IEEE, Arlington, VA, Dec 1983
- 85 Wegner, P. 'Capital-Intensive software technology' *IEEE Software*, Vol 1(7) (July 1984), p. 7-45
- 86 Wiener, R.; Sincovec, R. *Software engineering with Modula-2 and Ada*, John Wiley & Sons, Inc., N.Y. 1984
- 87 Willis, R.R. 'AIDES Computer Aided Design of software Systems - II' (Hughes, CA), in *Software Engineering Environments* ed H. Hünke, North-Holland Pub Co., 1981 (Proc Symposium in Lahnstein, Germany June 1980), p 27-48
- 88 Yourdon, E.; Constantine, L. *Structured Design* Prentice-Hall, Englewood Cliffs, N.J., 1979
- 89 Zave, P. 'The operational vs the conventional approach to software development' *CACM*, Vol 27(2), (Feb 1984), p 104-118

Appendix A

Software Engineering Environments Index

This index provides additional information on the environments that are presented in the second chapter. It can be used in conjunction with Table 2.1 and the bibliography for a more complete description.

- AIDES_____ : ⊗ Automated Interactive Design and Evaluation System. Supports structured design.
Hughes Aircraft Comp., Fullerton, CA
ref. in : [87], described in sect. 2.1.4, p. 30
- Alvey_____ : ⊙ Alvey Project. A fifth generation project. Supports ECLIPSE
British Telecom, U.K.
ref. in Electr. Intl. May 1984, two pages overview.
- APSE_____ : ⊙ Ada Programming Support Environment.
DoD, USA
ref. in : [9], described in sect. 2.1.2, p. 20
- Arcturus_____ : ⊗ An SDE
Univ. of California, Irvine, Ca.
ref. in : [77], described in sect. 2.1.5, p. 32
- ARGUS_____ : ⊗
Boeing Comp. Services, Seattle, Wa.
ref. in : [74], described in sect. 2.1.5, p. 33
- CADAS_____ : ⊙ Rapid prototyping of embedded systems
ESG/FEG, München, Germany
ref. in : [8], described in sect. 2.1.5, p. 33
- CADES_____ : ⊗ Computer Aided Development Evaluation System. Structural modelling, in operating systems. Supports programming-in-the-large and had been used in the development of the VME/B operating system.
International Computers Limited, Kidsgrove, UK.
ref. in : [71] and [60]
- CAEDE_____ : ⊗ Carleton Embedded systems Design Environment.
Dept. of Sys. and Comp. Eng., Carleton Univ., Ottawa, Canada
ref. in : [6], described in sect. 3.2, p. 46
- Cedar_____ : ⊗ A Mesa-like language support environment. Resemblance with Smalltalk and INTERLISP
XEROX, PARC, Ca.
ref. in : [79], described in sect. 2.1.3, p. 24
- CHI_____ : ⊗ Knowledge based.
Kestrel Institute, Palo Alto, Ca.

⊙ : only specifications.

⊗ : product or prototype in use.

22
OF / DE

Figure B.2: The Ada program

```

task BUFFER is
  entry GET (C out CHARACTER),
  entry PUT (C in CHARACTER),
end;

task body BUFFER is
  POOL_SIZE constant INTEGER = 100;
  POOL array(1 POOL_SIZE) of CHARACTER;
  COUNT INTEGER range 0 POOL_SIZE = 0;
  IN_INDEX, OUT_INDEX INTEGER range 1 POOL_SIZE = 1;
begin
  loop
    select
      when COUNT < POOL_SIZE =>
        accept PUT(C in CHARACTER) do
          POOL(IN_INDEX) := C;
        end;
        IN_INDEX := IN_INDEX mod POOL_SIZE + 1;
        COUNT := COUNT + 1;
      or when COUNT > 0 =>
        accept GET(C out CHARACTER) do
          C := POOL(OUT_INDEX);
        end;
        OUT_INDEX := OUT_INDEX mod POOL_SIZE + 1;
        COUNT := COUNT - 1;
    or
      terminate;
    end select;
  end loop;
end BUFFER;

```

The producing task may contain the statements :

```

loop
  -- produce the next character CHAR
  BUFFER.PUT(CHAR);
  exit when CHAR = ASCII.EOT;
end loop;

```

and the consuming task may contain the statements :

```

loop
  BUFFER.GET(CHAR);
  -- consume the character CHAR,
  exit when CHAR = ASCII.EOT;
end loop;

```

(Source : The Ada reference Manual.)

- MELBA_____ : A system for the design of telecommunication systems. Supports automatic code generation (CHLL) from Module Structure Diagrams (MSD) and SDL.
ref. in : [23], described in sect. 2.1.5, p. 34
- MML_____ : ⊗ Multi-Microprogramming Line
Università di Bologna, Italy
ref. in : [3], described in sect. 2.1.3, p. 29
- Omni_____ : ⊗ A high level screen oriented programming environment that supports problem solving by tool selection and tool composition.
Virginia and Purdue Universities.
ref. in [1]
- PAISLey_____ : ⊙ RT systems, executable specifications and performance constraints.
AT&T Labs
ref. in : [89]
- PECAN_____ : ⊗ PSE providing multiple views.
Brown Univ., Providence, RI
ref. in : [66], described in sect. 2.1.3, p. 28
- Pegasus_____ : ⊗ Supports programming-in-the-large and facilitates the reuse of software. The focus is on design understanding.
University of California, Irvine
ref in : [42]
- PegaSys_____ : ⊗ Computer Science Laboratory, SRI International
ref in : [55] [56], described in sect. 2.1.4, p. 30
- PDS_____ : ⊗ Program Development System
Harvard Univ., Cambridge, Mass.
ref. in : [11], described in sect. 2.1.3, p. 24
- Prometheus_____ : ⊙ An interactive environment for the development and parallel execution of functional programs.
Virginia and North Carolina Universities.
ref. in : [2], described in sect. 2.1.5, p.34
- PWB/UNIX_____ : ⊗ Programmer's WorkBench
Bell Labs, Ill.
ref. in : [53], described in sect. 2.1.2, p. 18
- SDem/SDSS_____ : ⊗ Software Development Engineering Methodology / Software Development Support System. A system based on a project library, a module description language and a set of support tools.
Fujitsu Lmt'd., Tokyo, Japan
ref. in : [57] and [58].
- SDS_____ : ⊙ System Development System.
TRW, Defence and Space Systems, Al.

- SIMPOS_____ : ⊗ Sequential Inference Machine's Programming and Operating System. A fifth generation project
FGCS (ICOT, Mitsubishi, Oki, Matsushita), Tokyo, Japan
ref. in : [82].
- Smalltalk_____ : ⊗ The Smalltalk-80 environment.
XEROX, PARC, Ca.
ref. in : [29], described in sect. 2.1.2, p. 19
- SPP_____ : ⊙ Software Productivity Project : Embedded Systems, requirements, specifications
TRW Defence and Space Systems Group, Ca
ref. in : [21], described in sect. 2.1.2, p. 21
- SWB_____ : ⊙ Software WorkBench
A software factory supporting production control, project management, and application specification, in addition to the implementation and maintenance. Toshiba Corp., Tokyo, Japan
ref. in : [50]
- UNIX™_____ : ⊗
Bell Labs., Ill
ref. in : [83], [53], [43], described in sect. 2.1.2, p. 18
- USE.IT_____ : ⊗ A prototyping environment It is based on a specification language called Axes (written in C) supports system specification in Axes, analysis of specification correctness, and automatic generation of executable code from the specifications.
High Order Software, Cambridge, Mass.
ref. in : [34].

Appendix B

Example Program Listings

Figure B.1: The Harmony/C program.

```
#include "/h/harmony/kernel/include/sys.h"
#include "/h/harmony/kernel/include/rtclock.h"
#include "/h/harmony/kernel/include/templates.h"
#include "/h/harmony/kernel/include/sios.h"
/*-----
 *      This is a Harmony/C multitasking program
 *      written for demonstration purposes only. It consists
 *      of three user defined tasks, and shows the use of a
 *      simple bounded ring buffer.
 *
 *      steli@systems.carleton.cdn , June 1986
 *-----
 */

#define AOIS274 19
#define BUFFER 20
#define CONSUMER 21

#define GET 0
#define PUT 1
#define FAIL 1
#define DELAY 500
#define SIZE 10

extern main();
extern Buffer();
extern Consumer();
extern _Directory();
extern _Gossip();
extern _Sios_server();
extern _Tti();
extern _Tto();
extern _Serial_int();

extern struct SIOS_DATA sios_data;
unsigned buffer, consumer;
unsigned _Pnumber = 0;
struct TASK_TEMPLATE _Template_list[] =
{
    { 1,    main,          500, 7, 0 },
    { 2,    _Directory,    334, 7, 0 },
    { 3,    _Gossip,       292, 7, 0 },
    { BUFFER, Buffer,       200, 7, 0 },
    { CONSUMER, Consumer,  200, 7, 0 },
    { AOIS274, _Sios_server, 1000, 5, 0 },
    { TTI,   _Tti,         500, 2, 0 },
    { TTO,   _Tto,         500, 2, 0 },
    { 0,     0,            0, 0, 0 }
},
```


Figure B.1 (cont'd)

```

void (*(_Intvec[7]))() =
{
    0,
    0,
    _Serial_int,
    0,
    0, 0, 0
};

struct BUF {
    short int MSG_TYPE;
    short int MSG_SIZE;
    char Ch;
};

main()
{
    char *str = "Test string.";
    struct BUF request, reply;

    sios_data.RPLY_LEN = sizeof(struct SIOS_DATA);
    _Server_create (AOIS274, &sios_data);
    _Sel_output ( _Open ( "AOIS274:", WRITE ) );
    _Sel_input ( _Open ( "AOIS274:", READ ) );

    _Putstr ( "\n main creates the Buffer, " );
    _Flush();
    buffer = _Create (BUFFER);

    _Putstr ( " and the Consumer, \n" );
    _Flush();
    consumer = _Create (CONSUMER);

    _Putstr ("\n and then holds a Producer position.\n");
    _Flush();

    for (;;)
    {
        while ( *(str) != '\0' )
        {
            request.MSG_TYPE = PUT;
            request.MSG_SIZE = sizeof(request);
            request.Ch = *(str++);
            reply.MSG_TYPE = FAIL;
            reply.MSG_SIZE = sizeof(reply);
            while (reply.MSG_TYPE)
                _Send ((char *)&request, (char *)&reply, buffer);
        }
        str = 0;
    }
}

```

Figure B.1 (cont'd)

```

Buffer()
{
    int      inx, outx, count;
    char     buf[SIZE];
    unsigned requestor;
    struct   BUF request, reply;

    inx = outx = count = 0;

    for (...)
    {
        request.MSG_SIZE = sizeof(request);
        requestor = _Receive ( (char *) &request, 0 );

        switch (request.MSG_TYPE) {
            case PUF : if ( count == SIZE )
                        reply.MSG_TYPE = FAIL;
                        else {
                            reply.MSG_TYPE = 0;
                            buf[inx] = request.Ch;
                            inx = (inx+1) % SIZE;
                            count++;
                        };
            case GET  : if ( count > 0 ) {
                        reply.MSG_TYPE = 0;
                        reply.Ch = buf[outx];
                        outx = (outx+1) % SIZE;
                        count--;
                        } else
                            reply.MSG_TYPE = FAIL;
        }
        reply.MSG_SIZE = sizeof(reply);
        _Reply ( (char *) &reply, requestor);
    }
}

```

Figure B.1 (cont'd)

```

Consumer()
{
    int    delay;
    char   *rstr = "Received str";
    struct BUF request, reply;

    _Sel_output ( _Open ( "AOIS274 ", WRITE ) );
    _Sel_input  ( _Open ( "AOIS274.", READ ) );

    for (...)
    {
        request.MSG_TYPE = GET;
        request.MSG_SIZE = sizeof(request);
        reply.MSG_TYPE = FAIL;
        reply.MSG_SIZE = sizeof(reply);
        while (reply.MSG_TYPE) {
            _Send ( (char *)&request, (char *)&reply, buffer);
            if (reply.Ch != '\0')
                *(rstr++) = reply.Ch;
            else {
                _Putstr(rstr)
                _Putstr("\n");
                _Flush();
            }
        }

        while (delay++ < DELAY);
    }
}

```

Figure B.2: The Ada program

```

task BUFFER is
  entry GET (C out CHARACTER),
  entry PUT (C in CHARACTER),
end;

task body BUFFER is
  POOL_SIZE constant INTEGER = 100,
  POOL array(1 POOL_SIZE) of CHARACTER,
  COUNT INTEGER range 0 POOL_SIZE = 0,
  IN_INDEX, OUT_INDEX INTEGER range 1 POOL_SIZE = 1,
begin
  loop
    select
      when COUNT < POOL_SIZE =>
        accept PUT(C in CHARACTER) do
          POOL(IN_INDEX) := C;
        end,
        IN_INDEX := IN_INDEX mod POOL_SIZE + 1,
        COUNT := COUNT + 1,
      or when COUNT > 0 =>
        accept GET(C out CHARACTER) do
          C := POOL(OUT_INDEX),
        end,
        OUT_INDEX := OUT_INDEX mod POOL_SIZE + 1,
        COUNT := COUNT - 1;
    or
      terminate;
    end select;
  end loop;
end BUFFER;

```

The producing task may contain the statements :

```

loop
  -- produce the next character CHAR
  BUFFER.PUT(CHAR);
  exit when CHAR = ASCII.EOT;
end loop;

```

and the consuming task may contain the statements :

```

loop
  BUFFER.GET(CHAR);
  -- consume the character CHAR,
  exit when CHAR = ASCII.EOT;
end loop;

```

(Source : The Ada reference Manual.)

Appendix C

Abbreviations and Acronyms

ADT : **Abstract Data Type**
 CACM : **Communications of the ACM (Association for Computing Machinery)**
 CAEDE : **CArleton Embedded System Design Environment**
 CCITT : **Consultative Committee on International Telephone and Telegraph**
 CHLL : **CCITT HLL**
 DSE : **Design Support Environment**
 HLL : **High Level Language**
 IPSE : **Integrated PSE**
 NCC : **National Computer Conference**
 PDL : **Programme Description Language (Ada[®] PDL)**
 Proc. : **Proceedings**
 PSE : **Programming Support Environment**
 RT : **Real-Time**
 SDE : **Software Development Environment**
 SEE : **Software Engineering Environment**
 SDL : **Specification and Description Language or System Descriptive Language (in CADES)**
 SDWA : **System Design With Ada**

END

1 710 4186

FIN