12-1993

Investigation into Formalization of Domain-Oriented Parallel Software Development

Daniel P. King

INVESTIGATION INTO FORMALIZATION OF DOMAIN-ORIENTED PARALLEL SOFTWARE DEVELOPMENT

THESIS

Daniel P. King
Captain, USAF

AFIT/GCE/ENG/93D-08

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/93D-08

INVESTIGATION INTO FORMALIZATION OF DOMAIN-ORIENTED PARALLEL SOFTWARE DEVELOPMENT

THESIS

Daniel P. King
Captain, USAF

AFIT/GCE/ENG/93D-08

AFIT/GCE/ENG/93D-08

Investigation into Formalization of Domain-Oriented Parallel Software Development

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Daniel P. King, B.S.E.E.

Captain, USAF

December 15, 1993

Acknowledgements

They say the road to wisdom is a long and arduous path with many *alternative* paths and potholes - but let's just say they're different and various. Hopefully the difficult journey taken to complete this Master's thesis has yielded some wisdom, but all I know is that this document is finally done! There were many pitfalls to get to this point and I have many people to thank for helping me through the rough times.

First, and foremost, I must thank my advisor and committee. Major David Luginbuhl was always there to help when I stopped making sense and without him this thesis would not parse. Thank you for helping me get this document into shape!

Dr. Gary Lamont has provided me with constant guidance throughout this process. He inspired me and challenged me every step of the way. Through his insights, I have gained a greater appreciation for and understanding of computational theory.

Major Paul Bailor helped to narrow the research focus and provided me clarity when I needed it most. His guidance through the Knowledge Base Software Engineering Group helped organize the issues.

I'd also like to thank my classmates from both the Pagan (PAAG) and KBSE groups for helping to answer the many questions that arose. Special thanks to: Vincent Droddy the wise mentor of the Pagans and scribe of the Pagan Compendium bible, Russell Warner and Jay Cossentine — the ones with all the answers to any computer questions, and Mark Gerken for helping in my literature search for knowledge-based systems. And all the rest who lent a hand or kind word when I needed it most.

Although my set of constructs may be complete, my journey for wisdom is not. I will finish my journey when Achilles catches the Tortoise! And if asked a question on this subject, I will merely reply (in true Zen fashion), "Mu"!

Daniel P. King

Table of Contents

List of Figures

Figure Page

Abstract

The transformational programming paradigm provides a formal design and implementation process that can be semi-automated to assist in software development. Formulated properly, the transformational programming techniques preserve the correctness of the transformations (mappings) between the various levels of development. Through stepwise refinements, the problem specification is transformed from a high-level abstract solution description to a detailed implementation. The burden on the software designer can be reduced by formalizing not only the transformation process, but also the domain-specific syntax used in representing and solving the problem. Current transformational models, however, do not include the ability to adequately represent concurrency. This research investigates a design process for bridging the gap between the informal and formal problem descriptions by applying formalized concurrent domain-oriented knowledge and software architecture concepts. Specifically, it investigates the application of concurrent software architecture models to structure the transformation process and facilitate the mapping to unique parallel implementation architectures. Domain-specific and software design knowledge interact within the transformational development process in the creation of an automated implementation. A semi-automated environment based on a complete set of proposed parallel composition operators is considered. A description of this environment is presented as applied to semi-automated parallel software development using transformational techniques and domain-specific knowledge.

Investigation into Formalization of Domain-Oriented Parallel Software Development

I. Introduction to Formalization of Domain-Oriented Automation

1.1 Background

The realization of a computational model takes a combination of many disciplines of mathematics and science. Various logical models and algebraic theories (represented in software) are combined with electrical device technology (realized in hardware) to create a logical symbolic manipulation device (the digital computer). The computer's operation reflects a logical process which appears to represent a form of a general purpose *intelligent machine.* Whether the *intelligence* is embedded in the machine hardware or in the software or in the meaning of the symbols (semantics vs. syntax) is the subject of debate and the basis for research into artificial intelligence (60). However, the computer represents a means to automate well defined processes and tasks through the design of an interacting software and hardware model (computational model).

Problem solvers capture real world problems within a mathematical/logical model and map this to a computational language description (a representation or encoding of the problem solution). The execution of the computational language representation operates through the symbolic manipulation of a syntactical model based on a set of logical instructions. The syntactical model is defined (at some level) by the logical processes syntactically encoded in the language representation. With specific domain inputs, the computer hardware generates the appropriate output signals based on this executing syntactical model. The syntactical model is related to the computer hardware through the compiler mapping(s) or transformations between the high-level computational language encoding and the low-level machine language code. The computer hardware execution responds to the series of machine code instructions by exhibiting specific behaviors or information

transfers. The meaning (semantics) of the computational model is reflected in the overall automated process through the human interpretation of the syntactical model, the input signals, and the output signals or a device response to the output (control) signals.

Traditional software development consists of encoding a problem solution, through an informal design process, into an executable language; i.e., a language with a direct compiler mapping into a machine executable form. General software design and implementation require the encoding of the high-level conceptual model of the problem solution into the syntax of the desired computer compiler language. The creation of the conceptual model implies an informal or partial mapping between the informality of the natural language description of the problem, its solution, and the formality of the language constructs employed. A formalized software development process requires that this semantic gap be breached. The method for bridging the semantic gap (between one syntactical representation and another) defines the development process.

Unfortunately conflicts exist in the various goals of software development. The designer evaluates the high-level problem specification and solution specification based on *understandability* – an intuitive semantic relationship between the resulting solution specification and the problem specification. On the other hand, the designer evaluates the lower-level design specification, the program (solution's implementation), based on *efficiency* – reduced or bounded execution time and memory space. Both representations are evaluated based on transformational *correctness*– a mapping to the desired problem behavior. The designer requires some formal means to *bridge the gap* between the high-level specification abstractions and a hardware executable language encoding.

This research deals with parallel software development through transformation. The proposed development process incorporates concepts of domain-oriented design and parallel software architecture modeling. In the remainder of this section, a general description of the software development process, software architecture modeling, domain-oriented software design, and parallel software design is provided. The purpose of this description is for the reader to understand the

nature of the problems and issues of parallel software development. A more detailed discussion of these background topics is provided in later sections of this document.

1.1.1 Software Development Process. The underlying problem of software development, even in the more structured processes of today, is the capturing of a design representation of the application problem in a syntactical form for which there exists a relation to the desired problem's behavior model. The question the designer needs to answer in order to achieve this mapping is, *Does there exist a direct relationship (design) between the concepts and syntax of the problem domain and the constructs and syntax of the language chosen?* The answer to this question evolves through the creation and selection of data and control structures to be encoded in the selected compiler language syntax.

Most modern programming methodologies apply concepts of *encapsulation* and *abstraction* (27). The software designer abstracts and encapsulates properties and concepts from the application into a form of abstract data type (ADT). The software designer creates a control structure (or algorithm) for the interaction of ADT's and encodes this within a computer language's syntax. The designer transforms and refines this syntactical model into an executable form using a design process. The design of the syntactical model and its mapping to the hardware determines the efficiency and correctness (effectiveness) of the computational model. The process can be viewed as a *transformational model.*

In the software transformation model, a specification of the problem is transformed from a high-level specification solution version into the executable implementation version (representation). The transformational paradigm can use a *wide-spectrum* language for the various syntactical encoding of the problem. A *wide-spectrum* language includes both high-level and low-level constructs within the formal context of the language. The transformation between alternative syntactical representations can be formally defined within the language.

The transformational development process progresses through a series of refinements, generating a new specification version at each level or step (2, 41). For the various syntactical representations of the problem to maintain the desired correctness in their encoding, there must exist direct mappings between the various specification versions. The mappings relate the specification back to the desired problem behavior level. The creation of these mappings is a formalized development process in itself. Through a formalized process description the transformations can be automated.

1.1.2 Software Architecture Modeling. As software designers gain more knowledge about software development and domain-specific applications, the transformation knowledge can be structured and formalized into specific design methodologies and techniques. Researchers have proposed development methods that combine these design methodologies and concepts with specific mathematical and engineering modeling techniques (27, 36, 55, 78).

Designers apply transformations to abstract problem details and encode problem knowledge into specific hierarchical model descriptions. The designers use the models as intermediate tools for understanding the application problem solution (and its behavior). Specific properties of the problem's functional specification are abstracted and encapsulated into properties(or behaviors) of the solution model using different graphical representations (languages). Examples of the alternative development methods that combine modeling and software design techniques include: Object-Oriented Design (106), Requirements Modeling Language (55), and the Object-Connection-Update (OCU) model (77). Underlying each of these design methods is a specific modeling description language that defines the abstract architecture.

The design techniques can be applied to the creation of hierarchical syntactical models of design. One structures a syntactical model for an application based on the chosen model's description language and the specific modeling methodology. The structure of a model may be created using a specific *software (design) architecture* for the problem. The more sophisticated modeling methodologies create specific *architectural* views of the various problem solutions and have been

referred to as software architecture models. Specific software architecture modeling techniques can be used to structure the design through transformation processes and thus provide a hierarchical abstract description of the problem solution.

1.1.3 Domain-Oriented Software Design. The informal nature of the hierarchical transformations and encodings in early (and some current) software development make it more of an art form than a strict discipline. As designers apply more of an engineering discipline to the development process, and as they gain more of an understanding of the problem domain, a domain-specific structure and formalism can be developed. The software designer can capture the knowledge of the transformation process and the problem domain in an explicit logic knowledge base. The designer extends the domain-specific knowledge base through lessons learned (experience) creating new theorems and axioms. The designer can then use this knowledge in the transformation design process. Thus, the software design and domain knowledge interact in a structured manner within the software development process (13).

The designer encodes the acquired problem and solution knowledge into a domain model. A *domain model* consists of a structured collection of domain objects, operations and relationships, and lessons learned encoded as rules, guidelines, or functional transformations. The domain objects represent actual entities within the problem domain. The advantage of the domain model is the direct relationship of the objects to the domain constructs. The domain modeler represents each object as an abstract data type or specification schema, encapsulating the domain concepts and domain entities. The model abstracts the details through a structured hierarchy and reduces design complexity. The use of the domain model within the design process creates a domain-oriented approach to software development (98).

A reduction in design complexity may be attributed to the evolution of the design strategy from informal to formal as the designer gains more knowledge about a particular domain. Using a domain-oriented design process formalizes the knowledge about the domain-specific models. The

designer uses this formalized knowledge to create a specific domain grammar (formal model) for describing an application. The domain-specific knowledge can also be combined with software architecture modeling techniques to create domain-specific software architectures (DSSA) (75). The use of a domain-oriented design process reduces the semantic gap described previously.

1.1.4 Parallel Software Design. The goal in the use of parallel machines is to obtain speed-up through the efficient utilization of multiple processors applied to an application problem. When the additional factors for parallel considerations are added to the software design process listed above, the complexity of the process is increased. The designer must also capture the inherent concurrency of the problem solution in order to apply efficient optimizations and map the design specification to a target parallel platform. The appropriate synchronization and communication structures must be added to maintain behavior correctness. The concurrent control structures face new design problems not encountered in sequential systems (4).

The actual mapping of the decomposed tasks of the problem solution (logical processes) to the parallel processors (physical processors) in order to obtain maximum efficiency adds additional complexity to the transformation process. One reason for this complexity is that the mapping/scheduling problem for multiple processors is known to be NP-complete (see Appendix B) (48). In addition, the designer must consider tuning the problem model's task sizes to the grain size of the target platform to achieve a proper processor load balance. The alternative decompositions and numerous schedules to be considered drive the combinatorics of the overall design process (26).

The designer must decide where in the hierarchical design process to include parallel considerations. Most experts agree there should be a separation between the problem solution specification and the architectural requirements; however, the designer must at some point take into account the target platform for execution. The parallel environment requires additional approaches to the traditional methods of sequential software design. The inherent parallelism or concurrent behavior

must be captured at all levels of the software design. The designer must delineate the original problem solution specification in a method that minimizes the dependencies of the tasks in order to maximize possible concurrent execution (4, 23, 26, 80).

Following problem identification, the designer must optimize the design specification and map it to a specific implementation through a series of transformations. The designer must apply a method for mapping the logical tasks to the physical processors of a specific machine that obtains acceptable performance. The problem may require further transformations and restructuring in order to facilitate the mapping and obtain acceptable performance. The variations in the machines may also require a specific language and specific control structures (see Appendix C).

Parallel software design has been traditionally accomplished in two ways (7, 23, 50):

- Parallelizing Compilers. The parallelizing compilers can be used to automatically restructure sequential programs for parallel execution. The compiler uses dependency analysis and other heuristic algorithms to identify independent tasks. The parallel compilers are limited by the algorithms for identifying concurrency and have only achieved minimal success.
- Explicit language constructs. This method uses parallel language constructs to explicitly identify concurrency for parallel algorithm specification. The design analyses the problem to identify independent tasks in the solution specification. A lack of formalized methodology for specifying and designing parallel software limits the use of existing parallel languages .

Thus, the parallel environment is made difficult not only by the number of varying architectures, languages, and mappings, but also by the need to capture the inherent parallelism explicitly.

1.2 Problem

The advances in parallel hardware architecture designs have out-distanced and overwhelmed current software capabilities (45, 50). This situation makes it difficult for many software engineers to use the hardware platforms effectively and efficiently. What compounds this problem is that

each parallel architecture may have some unique programming requirements, therefore requiring unique programming tools, a unique programming environment, and a unique design specification.

Due to the software design gap between hardware and software, several authors have proposed automation, development, and modeling techniques, and the tools and systems environments for supporting the design techniques (10, 12, 14, 36, 39, 41, 55, 65, 75, 89, 99, 105, 106). Designers develop automated systems to ease the burden on the software designers, engineers, and programmers; however, the existing automated environments do not include sufficient levels of parallelism. A complete formal, integrated environment for parallel software development does not exist and a general model may not be possible because of the complexity issue.

This thesis investigates: *the conceptual development of a formal, integrated environment for parallel software development using concepts of transformational programming, domain-oriented design, and software architecture modeling.*

This thesis proposes a conceptual software development platform integrates various ideas proposed by other researchers, unifying them in a single model for the parallel environment. It describes an automation platform for the full spectrum of parallel software design through transformation processes (2, 72). The feasibility of constructing the environment is informally demonstrated by analyzing: the specification, modeling and design process; the parallel design process; formal specification languages and environments; and automated design methods. The emphasis of this thesis is on investigating the required constructs for explicitly specifying parallelism.

1.3 Summary of Current Knowledge

The theories of automatic program generation have been around since the first compilers of the 1950's. With each new generation of languages, the hierarchy of higher and higher level languages has grown, and compiler technology has evolved. Besides the use of the compiler as an automated tool, several researchers have proposed other automated assistance tools (25, 70, 116).

The technology thus far has not bridged the gap between the informality of natural language problem description and formal language design specification, and the feasibility of a fully automated environment is questionable (10). Current semi-automated assistance tools attempt to bridge the gap and reduce the development time (10, 12, 93). Designers also attempt to reduce the time required to generate new designs through reuse strategies (14).

The initial reuse concepts emphasized reuse of coded software modules. The designers created libraries of software components defined by a functional specification and an interface description. The disadvantage of this paradigm was the extensive search the designer was required to perform in order to find the appropriate component. The designer also required an extensive knowledge of the underlying taxonomy of components. Advances in database management systems have improved the reuse paradigm by creating a sophisticated data structure referred to as a *knowledge base.* Current reuse strategies concentrate on the reuse of the software development process' structured knowledge by reusing software models (14, 77).

Some researchers have combined an extensive inference engine and a knowledge base to create *Automated Transformational Programming Systems* (10, 12, 14, 89, 93). The transformational systems apply transformations to derive a more detailed specification. A description of the general transformational paradigm is provided in Figure 1.1 (2).

Existing system structures vary, depending on whether the initial input is a formal or an informal description of the problem. They also differ on the formality of the development process to generate an executable program. Because the systems make extensive use of a knowledge base, *knowledge-based software engineering* has become the common term for describing these development processes.

The transformational systems use alternative methods for performing a stepwise refinement of the specification. The alternative methods include: heuristic search techniques, rule-based synthesis and derivation, taxonomic hierarchies, or a combination of these techniques (12). The method used

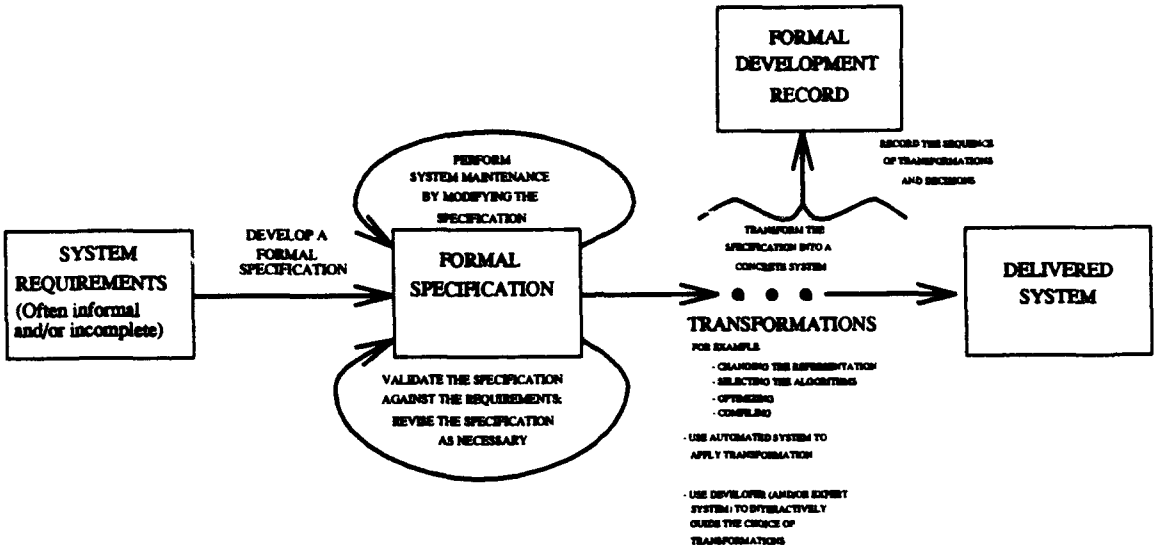

Figure 1.1 The Transformational Programming Paradigm

by the system determines the structuring of the information within the knowledge base. The transformational process may incorporate the synthesis or derivation of knowledge. The synthesis techniques derive new knowledge by using rule-based strategies (forward or backward chaining) and algebraic representations. The synthesized knowledge may be used to guide the transformation process. Automated systems that use these expert systems techniques may also be referred to as *Software Synthesis Systems* (12, 93, 116).

Although advances have improved the specification process, it remains difficult for designers to intuitively recognize a direct relationship between the formal language design specification and the problem description. Through the creation of a domain-specific grammar, an application specification can be generated with a more direct relationship to the domain problems.

The incorporation of domain-specific objects, operations, and transformations has been discussed by several researchers (10, 14, 40, 77). Many of these methods are limited in representing concurrency. To be effective, the inherent concurrency within the domain objects and their relationships must be explicitly captured in their composition and specification.

1.4 *Approach*

We propose a formalized process for software development in the parallel environment. We include the specification of a conceptual platform for the development of parallel software, shown in Figure 1.2 as modified from (109, 123). Once this conceptual framework is implemented, an application engineer can create a high-level domain-specific specification and transform it into an executable parallel software architecture, tailored for a specific parallel hardware platform.

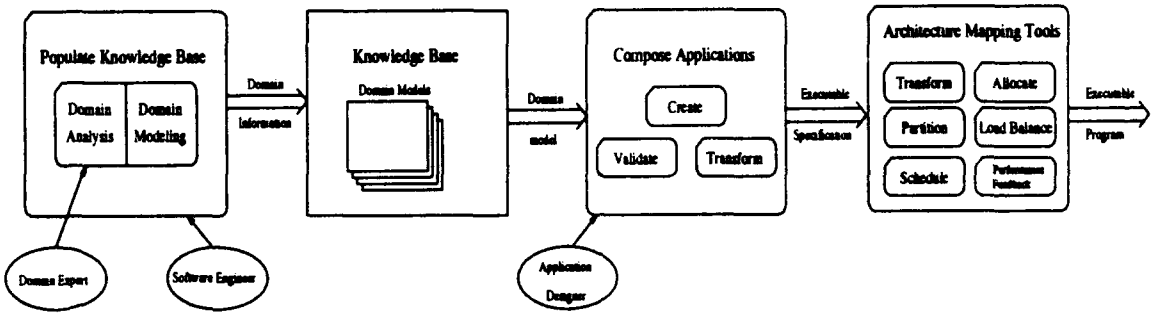

Figure 1.2 Domain-Oriented Parallel Software Environment

Our research extends the overall parallel design concepts into a platform extensively using the knowledge base in the development process. The encoded information in the knowledge base provides for the creation of a domain-specific grammar —generating domain-specific terms with a direct mapping to low-level constructs. The domain-specific grammar is written in an underlying wide-spectrum language. The domain objects and their operations are mapped to the functions and defined types in the language. The relationship between various objects creates a hierarchy within the knowledge base. The knowledge base additionally includes domain-specific transformations, rules, guidelines, and decomposition strategies to increase execution efficiency.

In order to generate and use the transformational model, a domain engineer must select and formally define both the domain language and the parallel software architecture model. A domain engineer and software engineer first capture specific information about the domain (domain knowledge) within a specification or model which includes parallel constructs. The software engineer

then creates an abstract syntax tree representation of the domain by parsing the domain model, generating a domain language in the process.

Using the created domain language and the various available tools of the platform, the designer composes a high-level specification (a syntactical model) of the application problem. The composition of the domain objects creates a software architectural model of the problem. The architectural model can take on various forms depending on the application domain, design strategy used (e.g. functional, object-oriented, logic, etc.), and architecture modeling description methodology chosen.

The designer can use various transformations to generate an executable design specification. The state-based representations of the design are embedded in the specification, and the constructs provide an *operational semantic*, allowing for execution. The execution of the design specification validates the formally specified behavior(s).

Once the designer validates the software architecture, the specification can be transformed using optimizations for the data structure, implementation language, and target machine. The tools of the environment can be used to assist the designer in transforming the high-level specification into a low-level version and mapping to a specific target parallel machine. The executing version is tailored for a specific target hardware platform and execution language. The designer may use various schemas to determine the appropriate transformations to apply when tuning the execution model for a specific parallel platform. Additional algorithms and transformations are available for the allocation and scheduling of the various tasks on the parallel architecture.

1.5 Scoping Assumptions

The goal of this research is to generally describe the issues involved with parallel software design and the application of formal methods, domain-specific design concepts, and transformational models to parallel design. The major focus of this research is on the required parallel language

constructs for specifying the inherent parallelism of an application as part of the overall conceptual platform. The definition of these constructs requires the analysis of current parallel architectures, parallel design techniques, and formal specification environments. A complete set of parallel constructs is defined based on the analysis of these topics; however, the completeness is only informally discussed and not formally proved.

We introduce a single general transformational model as a possible solution among several alternatives. The research presents a conceptual design; no implementation of the environment is attempted because of the time constraints. We use parallel merge sort and parallel search as general examples.

1.6 Document Overview

The reader of this document is assumed to have a general knowledge of software design, parallel design, and software specification and modeling. In addition, a basic understanding of computational theory is assumed. Several Appendices are included to provide additional detailed background in these areas.

- **Appendix A: Software Modeling, Specification, and Design**

  The software engineering concepts and design methodologies currently being used rely on specific properties and characteristics of the design models. This appendix describes several models and modeling techniques used in software engineering. The issues associated with problem identification and specification for using the alternative models are discussed. The evolution of modeling techniques applied to software engineering and their applicability to concurrency representation are also discussed.

- **Appendix B: Parallel Performance Issues**

  The overall objective of any parallel programming implementation exercise is to maximize performance. In order to accomplish this, the designer must understand the performance

issues. This appendix defines parallel performance measurement (e.g., speed-up, efficiency, utilization) and parallel software design properties (e.g., fairness, likelihood, choice). Also included is a description of specific factors that limit parallel performance.

- **Appendix C: Parallel Architectures**

  The implementation of a parallel program is dependent upon the selected hardware architecture, which may impact the software design. The parallel architectures have specific architectural characteristics. There are also alternative parallel processing models that impact parallel software design. This appendix describes the alternative parallel processing models (SIMD, SPMD, and MIMD) and alternative architecture models (multicomputer and multiprocessor). The appendix also discusses the applicability of these models to parallel software design.

- **Appendix D: Scheduling and Load Balancing**

  Task Management plays an important role in the design of parallel software. Specifically, the scheduling of the tasks and the load balancing of the system impact the strategies for mapping a software architecture to a target machine. This appendix describes in detail the scheduling and load balancing problems and methods used for attacking these problems. The scheduling and load balancing techniques provide informal methods that may be applied in creating the optimization transformations for the parallel software design environment.

- **Appendix E: Improved Task Management Using Graphical Methods**

  There are several techniques used for generating an allocation, schedule, and load balance for task management. However, the graphical methods approach to the task system model may provide a vehicle for combining software architecture modeling methods and architecture mappings. Extensive research was conducted into several techniques and tools that apply a graph-theoretic approach to architecture mappings. This appendix provides a description of the methods and tools investigated.

The thesis chapter are organized as follows.

- **Chapter 2: Background:**

  The first step was to evaluate and define the conceptual framework and the development process for using it in domain-specific software development. This initial research included an extensive evaluation of existing automated environments. A description of several of these environments is provided in this chapter. The Domain-Oriented Application Composition Environment being developed here at AFIT for the sequential programming environment is included. This previous AFIT research provided a basis for our approach (6, 72, 102). This chapter describes other background information on software modeling, specification, and design, and formal specification languages and environments.

- **Chapter 3: Methodology**

  Chapter 3 provides an overview of the parallel software design environment's conceptual platform. The general process, products, and support tools for parallel software development using this transformational programming model are described. A more detailed description of the research objectives is provided.

- **Chapter 4: Designing for Concurrency: Issues of Parallel Program Design**

  Since the overall design process is for the parallel environment, current issues in parallel design were studied. Concepts and techniques for parallel software and algorithm design are discussed. This chapter also describes methods for improved task management within the parallel design environment. Appendices D and E describe in greater detail specific tools and algorithms for applying a graph-theoretic approach for improved task management.

- **Chapter 5: Formal Specification Languages and Environments**

  The kernel to the conceptual environment is the design of a language and grammar from which the domain model is generated. The language describes the composition of the domain objects into a software architecture. The specification may require a wide-spectrum language

with high-level, low-level, parallel, and sequential constructs to support the various transformation capabilities and execution models. Several existing formal specification languages and environments are described and evaluated.

- **Chapter 6: Requirements for a Parallel Language and Environment**

  The languages and environments are evaluated based on the overall established framework for parallel design. This chapter describes the requirements for the conceptual platform and its parallel specification language. The languages and environments are evaluated based upon their formal syntax and semantics, and their concurrency features. An environment is selected for the analysis of a complete set of parallel constructs.

- **Chapter 7: Formal Definition of a Complete Set of Parallel Constructs**

  The conceptual parallel software development environment described in Chapter 3 requires a language with a complete set of parallel constructs. Chapter 7 provides a formal definition of the parallel constructs. This chapter provides a description of the formal syntax, a general discussion of the properties of the operators, their formal operational semantics, and their conceptual design and integration into the automated environment.

- **Chapter 8: Example Problems**

  Parallel applications for merge sort and best-first search are used to demonstrate the use of the parallel operators described. This chapter provides a general description of the application problems and their specification using the new operators.

- **Chapter 9: Results, Recommendations for Future Research, and Conclusions**

  This chapter summarizes the research, highlighting the results and conclusions, and pointing the way for future research. It summarizes the results of each chapter and applies these to a detailed description of the development process, products, and tools for the envisioned parallel design environment. This description is followed by specific conclusions and recommendations for future research.

1.7 Chapter Summary

The burden on the parallel software designer is reduced by formalizing not only the transformation process, but also the domain-specific syntax used in representing the problem. Our research investigates a design process for bridging the gap between the informal and formal problem descriptions by using formalized domain-oriented knowledge and parallel software architecture concepts. It investigates the application of software architecture modeling techniques with concurrent properties to structure the design process and facilitate the mapping to a parallel implementation architecture. The domain and software design knowledge interact within the automated development process to generate a more efficient and effective design.

The development environment's underlying language is the unifying feature of the environment. All transformations should occur within a wide-spectrum language. In order to capture the application's potential concurrency, the specification of the application must include parallel language constructs. The major focus of this research is defining the parallel language requirements for the proposed software development environment. The parallel constructs must not only support the specification of parallel application, but also their eventual mapping to a parallel architecture implementation.

II. Background

2.1 Introduction

In Chapter 1, we established that the development of software requires some method of conveying the problem to a designer and transforming the requirements into an executable version. For this process, the requirements of the problem need to be captured in a specification. Initially, this is an informal description of the problem. Through analysis and modeling techniques a greater understanding of the problem is gained and the designer attempts to formalize the description. Formal mathematical models and formal specification languages can assist the designer in the formalization of the problem description. The addition of an automated environment may provide the tools to assist the designer in transforming the formal problem description into an executable version.

The appropriate automation tools provide a software engineer an abstraction of the actual implementation environment and assist the engineer in the software development process. The tools abstract the details, enabling the engineer to concentrate on understanding the application problem and problem domain. The engineer may additionally use performance measurement tools to evaluate efficiency transformations and assist in mapping to a specific target platform. An understanding of existing automated tools, processes, and environments, and their applicability to transformational programming is required.

We will use the terms *designer* and *software engineer* interchangeably throughout this document.

This chapter is divided into five sections. This section provides an introduction to the chapter and an overview of topics covered. Section 2.2 describes the evolution of modeling techniques within software engineering. Various methods of modeling software systems are discussed. Alternatively, software designers also use formal specification languages to describe problems. Section 2.3 discusses the use of formal specification languages, providing an overview of several example languages. A

description of the categorization of languages by semantic description is also provided. The various models and specification languages provide a structured design process. The formal structure of the process allows for its semi-automation. Section 2.4 includes a discussion of the various software automation techniques. Descriptions of example tools and design philosophies proposed by current researchers are provided. Finally, Section 2.5 summarizes the chapter.

2.2 Models in Software Engineering

Software developers first used flow charts or pseudo-code to characterize software systems. The flow charts or pseudo-code expressed the sequence, selection and iteration of the processing steps for the system. However, the data flow, underlying performance, and concurrency were not appropriately described. Software developers also used program structure charts to capture the data and control flow, but concurrency and performance were still not captured by these models. Multiple models were required to represent a complete problem view — the static and dynamic behaviors, the data and control structures, and the data and control flow. Mappings between alternative models for a uniform view of the problem was difficult at best.

As the designer applied more of a mathematical discipline, models incorporated concepts for mathematical functions, abstract data types, functional hierarchy, control flow theory, and graph theory (5). Specific structuring techniques were defined within the models for encapsulation based on objects, functions, domain-specific concepts, and software architecture concepts. The structuring of the models allowed for aspects of the model to inherently represent certain properties, embedding specific knowledge within the model's structure.

Appendix A provides a detailed description of the issues involved with problem modeling and problem identification. Section A.2 provides an overview of the evolution of modeling methods, their relation to specification approaches and theories, and how problem identification impacts modeling methods. This section describes the evolution of modeling techniques within software

engineering. Alternative methods of modeling software systems to capture the functional, state, functional hierarchy, and graphical aspects of the application are discussed. A reader not familiar with these topics may reference Appendix A before reading this section.

This section provides an overview of several of the important and more sophisticated modeling concepts currently being applied in software engineering. Specifically described are: object-oriented models, Petri nets, task system modeling, software architecture modeling, domain-oriented modeling, and the Object-Connection-Update Model. The descriptions outline the structure and formality of the different modeling methods applied to software engineering. They also show how the different techniques can be used to represent certain characteristics of the problem space. In addition, Section 2.2.7 describes an approach for applying software models proposed by Iscoe in (65). His approach combines the concepts of object-oriented and domain modeling with an automated environment and knowledge base.

2.2.1 Object-Oriented Modeling. Object-oriented design dominates today's software development. Object-oriented design focuses on the objects or entities of a system, their relationships, and their interactions. Designers specify the program modules and data structures using objects. The designers use models to capture the objects' static, dynamic, and functional behaviors. The general properties of the objects are organized into *object classes.* The object classes provide common links between the three models for characterizing aspects of behavior. There are several references available for a more detailed discussion of object-oriented design (19, 80, 90, 106).

The static model (object model) displays the object classes and their static relationships, attributes, and operations. The dynamic model displays the dynamic behavior for each important object class using a state diagram. Finally, the functional model depicts a data flow between the objects and operations of the system.

The concept of abstract data types (ADTs) is predominant in the object-oriented approach, since an object class may alternatively be viewed as an ADT. An ADT classifies a type using

a formal description of the objects, operations, and axioms. The fundamental difference between ADTs and the object class of object-oriented design is that using object-oriented design, the designer only captures the axioms and additional constraints between the object classes in an informal way. These properties are captured using a data dictionary or by adding comments to the graphical models.

2.2.2 Petri Nets. The concepts of network flow and control flow have evolved significantly over the last decade. One reason for their improvement is the introduction of Petri nets. There are several variations of Petri net representation, with modifications designed to highlight certain features of specification. Thus, this section only provides a general description of the Petri net model. More complete descriptions of Petri nets can be found in (23, 97, 126)

A Petri net provides the notation for the formal representation of sequencing and concurrency through a comprehensive description of a system's control flow. A Petri net consists of a modified directed graph containing two types of nodes, places and transitions (see Figure 2.1). The place, represented by a circle or a box, defines locations for possible placement of tokens. The places are linked by arcs to transitions. A transition, represented by a bar, defines the possible movement of the tokens. The condition of the transition must be satisfied in order for the token to be moved. The transition contains both a pre-condition and a post-condition. The pre-condition may check for the existence of a token on the predecessor node, while the post-condition may specify the movement of the token to subsequent nodes. The token represents a possible control flow through the system. Through multiple tokens, concurrent control flow can be represented.

Petri nets can represent many interesting and necessary properties of concurrent systems. The flexibility of the control flow representations allows the specification of such properties as sequence, selection, concurrency, and synchronization. Properties such as freeness from deadlock and correct termination can be verified through the use of Petri nets. However, the unmodified Petri net concept expresses only control flow; data flow is ignored. As noted by Alford, even though

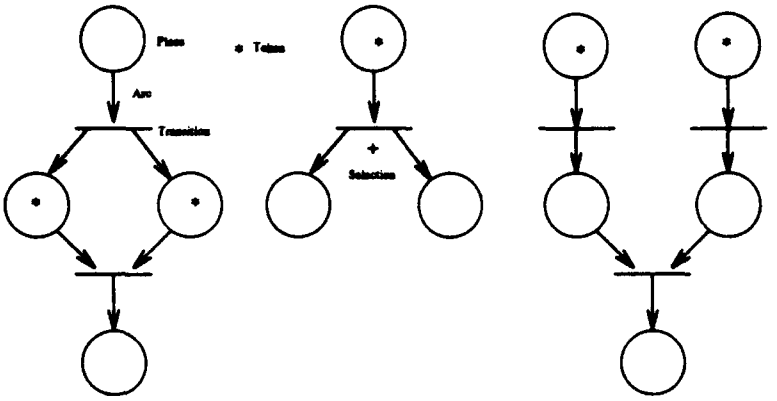

Figure 2.1 Petri Net Example

the transition conditions can be specified in terms of data values, the semantics for changing or saving data variable values is not an inherent part of the Petri net model (5).

2.2.3 Task System Modeling. The task system uses a graphical modeling technique for its representation, taking advantage of its power of expression for both the data and control flow features. Each node represents an execution task of the system, with the task encapsulation derived from the software design method used (e.g., object-oriented or functional). The designer connects the nodes in a specific system structure, providing a framework for its software architecture. The structure provides a representation of the precedence and sequence requirements of the system. A hierarchy of tasks may exist. In addition, the various tasks may be specified using domain-specific constructs, or the model may be composed in a domain-specific architecture.

The task system graph includes an *architecture graph* and a *task graph.* The *architecture graph* captures the properties of the hardware architecture of implementation. Each node of the graph represents a physical processor, with arcs representing a communication channel between processors. The weight labeling of the arcs may signify the communication time between the processor nodes or the communication bandwidth. The model may also make the assumption that each communication hop is of uniform time to reduce the complexity in generating a mapping.

The *task graph* represents the application problem. Each node represents a specific *task* for the application. The tasks may be encapsulated based upon an object-oriented or functional approach. The nodes include a weighting for the execution time of the task. Depending on the variation of task graph used, arcs may represent task communication interaction (e.g. in a Task Interaction Graph) (20), or data flow and data dependency (e.g. in a Static Macro Data flow Graph) (51).

Figure 2.2 shows an example architecture graph for an 8 node hypercube architecture. Note that the graph is a planar, undirected graph with uniform communication. Figure 2.3 depicts an example Macro Data flow Graph. A variation on a data flow graph is used for this representation. The directed arcs establish the data flows and data dependencies, and the nodes represent the logical processes. Figure 2.4 shows a Task Interaction Graph (TIG), where each undirected arc represents a requirement for task interaction.

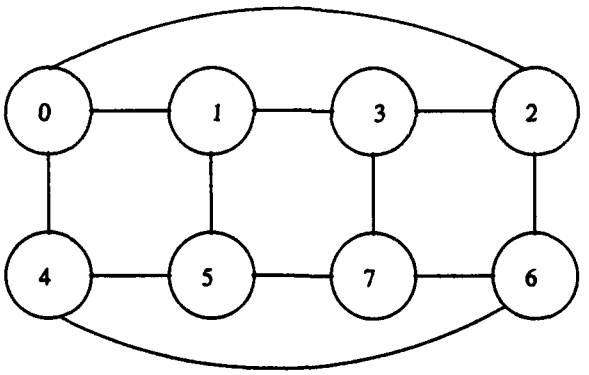

Figure 2.2 Architecture Graph Example

A complete task system graph creates a mapping between the architecture and task graphs. The mapping represents a scheduling of the tasks. There has been extensive research into this graphical method of task management. For a detailed discussion of the use of this graphical modeling approach, see Section 4.5, Appendix D, and Appendix E of this document.

The fundamental disadvantage of the task graph representation is its failure to adequately represent the dynamic control flow of a given application. The Task Graph method is primarily used for static problems, where the execution trace of the problem is known *a priori*. As more researchers

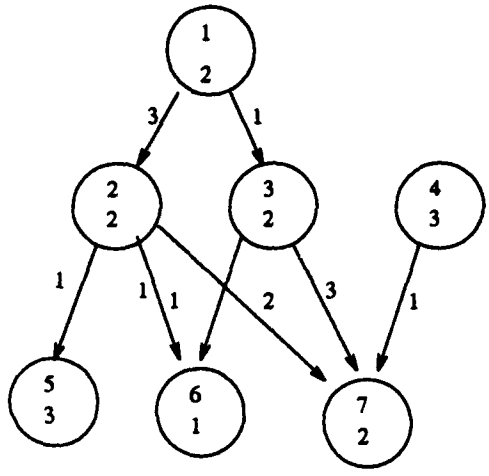

Figure 2.3 Static Macro Data Flow Task Graph Example

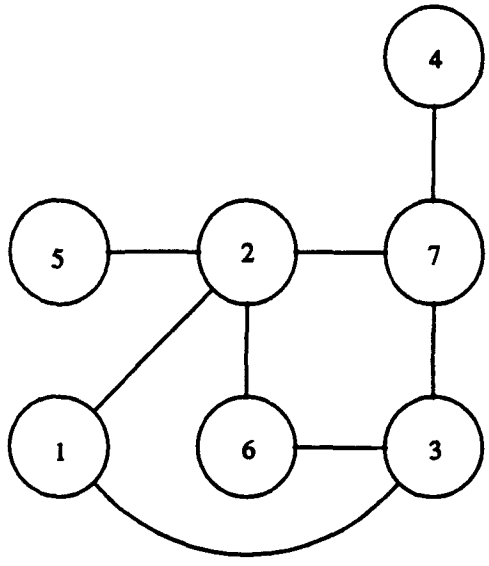

Figure 2.4 Task Interaction Graph Example

gain a greater understanding of dynamic partitioning, scheduling, and load balancing techniques, the use of the task system may be applied to model these dynamic applications. Examples of a dynamic problem include simulation applications, where the input data has a dramatic impact on the alternative data and control trace and, therefore, the solution.

2.2.4 Software Architectures. A software architecture is used as an abstraction of the properties that make up the interrelationship between the objects of a domain. The encapsulated objects are organized in a pre-defined structure or architecture. The organization of the logical constructs into an architectural format facilitates the mapping to a hardware architecture. There are various architectural models used in software design, providing alternative views of a problem. For example, a software architecture may incorporate concepts of a finite state machine.

Software architectures serve as frameworks for software reuse. The use of domain-specific software architectures (DSSA) was initially introduced by the Defense Advanced Research Projects Agency (DARPA) (75). Domain-specific software architectures serve as a common language in which domain engineers can discuss, understand, and teach the principles of their craft. The use of the term software architecture can be viewed as another means of describing some of the sophisticated modeling techniques used in software engineering.

By creating a formal way of specifying the interrelationship between objects within a domain or between two related domains, one can generate a hierarchical production process. Critical aspects of a domain or class of problems are identified. These critical aspects are then represented through specific constructs within a domain model. The designer creates a formal model of the architectural aspect of the domain. The interfaces between the various object primitives and subsystems (collection of primitives) are formally defined.

In applying these architecture compositions to a transformational model of software design, the designer can manipulate and combine domain objects using formally specified productions. He

can use these transformations to generate new applications by combining existing applications and extend existing applications by generating new subsystems from primitives (87).

2.2.5 Object-Connection-Update (OCU) Model. The Software Engineering Institute developed the OCU model as a method of applying their Model-Based Software Development approach to software development. The OCU model incorporates the concepts of software architecture into a specific modeling technique (77). The designer can specify an application problem using the features of the OCU model. The OCU model combines methods of functional, state-based, and graphical descriptions.

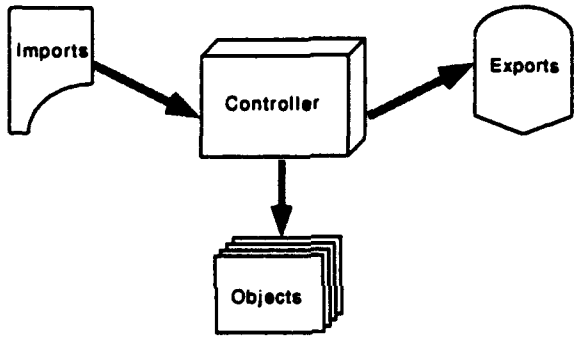

Figure 2.5 OCU Subsystem

The OCU model includes the following elements: controller, import area, export area, object, I/O device handler, surrogate (monitor and control), and executive (77). The designer configures the model into interactive subsystems controlled by an executive (see Figure 2.5) . The executive controls the interaction between subsystems and defines the system environment. Each subsystem includes a controller, a set of objects, an import area, and an export area. The objects models the behavior of real-world components. Each object has a *state* determined by the values of its attributes. The controller encapsulates the control structure of the subsystem, defining the dynamic behavior of the objects based on a given input. The primary function for defining the dynamic behavior (state transitions) of an object is the *update* function. Other functions are available for maintaining the objects, such as create, setstate, and destroy.

The OCU model provides an object-oriented structure while additionally specifying an explicit control structure. The model's structure supports layered design of a dynamic application. Each subsystem within the structure might be mapped to an individual processing element within a parallel machine through allocation and partitioning strategies. The executive and controller functions of the model define the interactions between the subsystems. These explicit control structures can be used to define the required communications, synchronizations, and execution precedence.

2.2.6 Domain Modeling. The concept of domain languages and domain-specific design has been the subject of several efforts (6, 11, 13, 39, 43, 64, 71, 72, 83, 98, 102). However, researchers have published little information describing the details of exactly what defines a domain language, specifically, how to partition and capture the various abstractions and behaviors of the domain. This structure becomes important when trying to characterize the software architecture. One possible reason behind the lack of such a generalized structure may be that the structuring itself is domain-specific.

Domain-specific information plays a major role in determining the design strategy. Designers need to focus on classifying the type of problem and various methods of finding a solution. The classifications would create a common vocabulary, possibly allowing for design knowledge reuse. There are several categories of general problems that have an extensive research foundation. These include search, sort, numerical analysis, data and control driven problems, and network flow. There additionally exist various methods or views for solution strategies, for example, divide and conquer, dynamic programming, deterministic search, evolutionary programming, linear and non-linear programming, and stochastic methods. How the designer identifies the problem determines the applicability of the various concepts and methods. All these factors impact the structuring of the domain model.

To define a domain, one must specify various aspects of it, including the axioms and constraints. All information about an application or problem domain is captured as *domain knowledge.*

As the domain engineer learns more about the problem domain, through the process of *domain analysis*, he adds the new information to the domain knowledge. The domain knowledge may be encapsulated and structured in a *domain model* or *domain specification.* The syntax used for the high level abstraction in the domain model becomes the *domain language.* The designer uses the domain language to specify a problem instance in an *application specification.*

Randour (102) provides useful insight into Prieto-Díaz (98):

> Rubén Prieto-Díaz defines domain analysis as the process used to identify, capture, and organize information. A domain language and a domain model are two of the outputs of this process (98:47-48). He further defines a domain-specific language as "a language with syntax and semantics designed to represent all valid actions and objects in a particular domain" (98:23) and "a collection of rules that relate objects and functions and which can be made explicit and encapsulated in a formal language and further used as a specification language for the construction of systems in that domain" (98:26). The domain language describes the objects and operations and becomes the framework for new specifications (98:23-24). The use of a domain-specific language provides a direct relationship between the application and the description language syntax, thus reducing the complexity and ambiguity of the description.

The problems of capturing, organizing, and representing domain-specific knowledge are further discussed by Wartik and Prieto-Díaz in (124). This paper presents an overview and evaluation of several methods of domain analysis by categorizing the existing methods. The authors describe how the different underlying factors of domain analysis impact the knowledge representation and categorization. The domain analysis techniques suffer from the same knowledge representation problems identified in Section A.2.2.

2.2.7 Object-Oriented and Knowledge-Based Approach to Specification Generation. Neil Iscoe's approach extends the object-oriented approach by using a comprehensive knowledge base. He applies domain modeling and domain-specific knowledge toward the goal of software reuse. His approach to software design uses a sophisticated object-oriented method of design. Iscoe defines the attributes for objects to be a unique name and set of values. When appropriate, he also defines stochastic values for measurement scale, unit and granularity; a set of population parameters; an initialization procedure; probability of change; and a state transition relation (66:33). Class

descriptions consist of a set of attributes, a set of functions used to create objects, a set of operations on the objects, a statistical summary, and a set of axioms or integrity constraints (66:77).

The statistical information, along with axioms and/or integrity constraints, adds a hierarchical layering of knowledge not found in other object-oriented approaches. This knowledge hierarchy adds a meta-level to the model and provides a framework for the incorporation of domain-specific information. Iscoe suggests applying domain knowledge to a meta-model or meta-language to instantiate a domain model. The meta-model or meta-language is a representation structure or language used to specify the knowledge about a given domain. The domain knowledge would provide rules, guidelines, and constraints for the composition of the object classes of the domain model, exhibited in the meta-level. The meta-level includes a formalization of the data dictionary features of other object-oriented approaches.

The formality of the description and ability to execute the model allow mathematical reasoning and inference to support design goals (65:301). Extensions to the process are provided through modifications to the meta-level, with automatic feedback through the operational capabilities. Iscoe defines a hierarchical decomposition of the classes that allows for specialized relations for inheritance and aggregation. The hierarchy provides an implicit domain structure (66:121). The designer can compose new classes from previously existing primitive classes using their composition (a denotational semantic), extending the hierarchy. Through this application of inheritance, we obtain a significant level of reuse of the domain structure.

Other researchers have emphasized the importance of using models in software engineering. In (36), D'Ippolito makes a strong argument for the application of sophisticated modeling techniques in software engineering. D'Ippolito asserts that the knowledge encapsulated in the various models can provide a potential for knowledge reuse. The model represents a "codified body of scientific knowledge and technology presented in a (re)usable form" (36:256). He argues that current research should investigate the use of more sophisticated modeling techniques (such as the OCU Model).

2.2.8 Summary. Sophisticated modeling techniques assist the designer in not only gaining a better understanding of the applications (its properties, characteristics, and behaviors), but also in encoding the problem within a syntactical representation with a direct mapping to a formal semantic. Combining mathematical concepts with the structured design concepts provides improvements to the design process through their logical hierarchy and formalization. Examples in logic, functional, and object-oriented design demonstrate the capabilities of the alternative modeling techniques. Domain-sp ific concepts and software architectural concepts provide an organization and structure that facilitates mapping between the problem and the computer.

The static and dynamic behaviors of the system, as well as the control and data flow, are captured in the various models used in development today. In addition, properties of concurrency can be explicitly or implicitly captured within a given model. The process describes the model in a modeling description language. In the more sophisticated techniques, the description language has a direct mapping to structures within the computer language. Thus, the model is encoded into the computer through a mapping between the modeling language and a computer language, creating a software architecture.

2.3 Formal Specifications

By formal language, we mean one whose semantics are based on mathematical formalism. A formal definition involves the use of a language that can be described using syntactic criteria, i.e., based on the form of the terms. Logics are formal systems that incorporate formal rules of reasoning, and whose semantics are clearly specified using mathematics (55). Many formal languages additionally include mathematical constructs within the language. Through the use of these mathematical constructs, axioms and theories can be generated within the language. The mathematical specification facilitates a method of logical proof for the well-formed formulas of the language.

The constructs of the language and productions should be precisely defined, allowing for a common understanding of their meaning. Many of today's computer languages can be formally mapped to first-order predicate logic (55, 56, 84, 60). The mapping provides the formality of the language and establishes equivalence relationships between comparable languages. As noted by Greenspan, defining a *formal semantic* provides a domain where meaning can be defined for properties of *consistency* and *correctness*. A language must be formally defined if one wants to have precise meanings for *well-formed* and *consistent*(55).

The languages used within computer science share certain aspects through a common mapping to mathematical principles. Researchers have attempted to analyze the languages to maximize their power of expression and efficiency of description. Fundamental theorems of predicate calculus and set theory appear throughout computer languages. Primitive type classifications of boolean and integers are available in all computer languages. More advanced structures can be specified using the primitive constructs. The extension of the language creates a taxonomic hierarchy of new knowledge (12). Unfortunately, the low-level mathematical terminology provides a complex environment for human interpretation and understanding.

By studying the properties and characteristics of the existing formal languages, we can gain a better understanding of the language requirements for a parallel software design environment. The remainder of this section includes a discussion of the semantic categorization of languages and an overview of the formal specification languages and environments evaluated in Chapter 5.

2.3.1 Semantic Categorizations. The *formal semantics* of various language models may be categorized through their operational, denotational, or axiomatic semantics (56).

An operational semantic describes the execution of a specification on a virtual machine. The interpretation of the objects and operations performed by the machine are given through a mapping to the machine's instruction set. The advantage of an operational semantic is the concrete and intuitive understanding of the operational environment provided to the programmer

and a direct feedback of the operational model. The model takes into account the environment, and constructs correspond to direct behaviors of the machine. The disadvantage is the machine-dependency, which ties interpretation to an implementation. The abstract nature of the language is lost with the machine-dependent constructs. One aspiring for true formality would require a proof of correctness of the machine (38, 122).

A denotational semantic uses a set of machine-independent operations on mathematical entities; all functions are defined mathematically, and state transitions are defined as mappings. A denotational syntax permits representation of error conditions as special elements in the functional range. Using a denotational syntax, one establishes the definition of new entities through a relationship to known entities. Since use of denotational semantics results in an abstract description not tied to an implementation, there is no machine-dependence. Since the resulting design consists of mathematical objects, the designer may perform a formal proof. However, the designer must be careful about ambiguity, under-specification, and over-specification. The designer must specify every possible eventuality for true correctness. Automated proofs are difficult and complex, often requiring higher-order functions and exhaustive search (38, 56).

Axiomatic semantics describe a program in terms of the functional relationship of inputs to outputs. The designer specifies the specific pre- and post-conditions, resulting in a characterization of *state*. The axioms are tied to the actual program, so there is no need to deal with generalities. There is also no need to define a mathematical function with all values of the domain, allowing partial function definitions. The design's soundness assures there is no need to specify error conditions. The proofs of correctness are conceptually simpler than other forms but may still require extensive search. The proofs should be based on sound mathematical theories, such as first-order predicate logic, lambda calculus, and temporal logic. On the other hand, it is difficult to define inference rules for the proof theory if arrays and pointers are involved. For many more complex data structures,

an expert may be required to define assertions with program statements. Axiomatic syntax is quite complex, making it difficult to automate (38, 56).

2.3.2 Examples of Formal Specification Languages. Examples of formal methods for description and modeling include UNITY (23), CSP (57), REFINE (103), GIST (10), CIP (54), PCN (44), PVM (49), Requirements Modeling Language (RML) (55), Petri nets (97), the Parallelation model (107), the Actors model (1),and Markov processes. Even such object modeling techniques as data flow diagrams and state transition diagrams (80, 106) can be used, since an object model is an abstract parallel architecture. Several of these languages are briefly described below.

Unbounded Nondeterministic Iterative Transformations (UNITY) combines first-order predicate calculus and temporal logic to provide a syntax for extracting the maximum parallelism from an algorithm. The purpose of UNITY is to form a foundation for parallel programming, which implies that it provides the smallest set of concepts needed for formally describing parallel programs. Unfortunately, UNITY is not executable due to properties of infinite execution of statements and lack of control flow (23).

Communicating Sequential Processes (CSP) provides a description of parallel computation based on a formal grammar description (58). Each process is viewed as a predicate. The processes are additionally characterized through a state description. The transition of states can be represented through the grammatical production rules. Because of its association to the methods of formal languages and grammars, CSP provides a powerful tool for modeling and analysis of processes. The abstract nature of CSP allows for the capability of equivalence mappings to other formal parallel representations. In (7), Bailor establishes the relation between CSP and Petri nets, whereas, in (114), Simmers evaluates the connection between UNITY and CSP. Like UNITY, CSP is not an executable language.

The REFINE programming environment provides design flexibility for the formal specification of sequential application problems. Structures for sets, sequences, functions, maps, rules, and

transformations are provided. The language is classified as a wide-spectrum language due to the inclusion of both high-level and low-level programming constructs. The framework of the language is currently build upon a LISP programming environment, providing for language execution capability. Through execution, one can verify the desired behavior characteristics for a given specification. Unfortunately, REFINE is not well suited for specification of concurrency (103).

GIST is oriented toward programs that manipulate dynamically changing data bases. GIST is a wide-spectrum language, containing both high-level specification constructs and constructs of existing low-level programming languages. GIST provides language constructs for the specification of problems through defining their objects, operations, and relations. It includes relations, temporal references, nondeterministic operations, and constraints (12). The use of *constraint* constructs allows the specification of integrity conditions for defining system behavior. Demons and agents are available for defining functional tasks and transformations (9, 94).

CIP-L provides specification and implementation constructs for both data and control structures. Included within the language are constructs for predicate logic, typed set operations, comprehensive choice, and general description. The description of the application is built using extensive mathematical principles. The designer must create the structures and functions used for application description through a recursive definitional process, where each new construct is defined based on existing functions and axiomatic types. Axiomatic types are defined through a formal schema definition approach. The approach is derived from algebra signature definition in mathematics and includes the definition of formal axioms (54).

Program Composition Notation (PCN) is a parallel design environment for developing and executing parallel programs (44, 24). One of the developers of UNITY, Chandy, was also involved in the development of PCN. Because of Chandy's involvement, the similarities between these languages are readily apparent. The PCN language formalizes several of the UNITY properties into an operational semantic. PCN, as the name implies, focuses on the parallel composition process.

A specific parallel composition operator is defined for composing blocks of code for mapping to multiple processors. In addition, a specific protected variable type is defined for communications and synchronization.

Parallel Virtual Machine (PVM) is a software package that allows a heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource. The purpose of PVM is to abstract the machine (computer) and its associative implementation control structure from the program designer (49). Unlike other design environments listed above, PVM is not a formal specification environment; however, PVM is a design environment providing a virtual environment for concurrent programming that allows for machine-independent design.

Each of the languages above has inherent formal properties through a formal language description and the inclusion of mathematical constructs for set theory and predicate calculus. A more complete discussion of the languages and their environments is provided in Chapter 5.

2.4 Software Automation

2.4.1 Introduction. This section defines specific terminology used in describing the various automation systems. It describes the transformational programming paradigm and the Transformational Implementation (TI) model proposed by Balzer. Section 2.4.2 describes several existing automation systems and techniques. For further reading on existing systems, see (10), (93), and (94). These sources provide a detailed description and evaluation of several available research and commercial systems and design techniques.

Researchers have been working on the extension of automatic programming concepts since the 1950's (10). The concepts combine research from compiler theory, artificial intelligence, and database techniques for a knowledge-based approach to software design.

The knowledge base provides a structure for the use of specific knowledge constructs in transforming, synthesizing, and generating software. Knowledge about the automation system's

task is usually explicitly encoded into the system's data structures. The system contains algorithms and mechanisms for retrieving, applying, and storing the knowledge. A variety of techniques and structures for building knowledge base systems have been developed. For example, various systems may use: heuristic search, rule-based methods, knowledge synthesis, or taxonomic hierarchies (10, 13, 12, 94). Barstow formally defines these techniques in (12).

Barstow provides the following classifications:

- *Heuristic Search* formulates the task as a space of alternatives that can be explored and compared.
- *Rule-based Methods* formulate the problem as a set of logical assertions and a set of special purpose inference rules.
- *Knowledge Synthesis* is a specialization of the rule-based methods.
- *Taxonomic Hierarchy* codifies the knowledge in terms of objects that are organized into a hierarchy of generalized classes.

The representation of the knowledge differs depending on the method chosen for the automated system. The techniques using a knowledge-based software engineering process may be classified as: Transformational Programming, Software Synthesis, and Program Generation. The classification into any of these design paradigms is somewhat arbitrary due to the varying categorizations and terminology used by the researchers. The distinctions are also obscured by the combination of philosophies used in designing some automated systems (93).

Transformational programming was described in Chapter 1. The next two paragraphs describe the other two classifications (specializations) of knowledge-based software engineering systems described in the literature.

Software synthesis is actually one style of transformational programming. It distinguishes itself from other techniques through its extensive use of theories stored in a knowledge base. The specification is viewed as an encapsulated model with a knowledge structure. The designer uses the knowledge to synthesize or infer new information, thus transforming the specification model. The synthesis enables the use of the inherent knowledge of the specification without the need for its explicit representation. Rule-based strategies of forward and backward chaining can be applied

to synthesize new knowledge. The knowledge can augment the existing specification (making the knowledge explicit) or be used to guide the synthesis process (12, 93, 116).

Program generation offers a strategy based on the reuse of software components. Various software components exist in a database for the designer's use. The components are categorized through their functional specification. The designer can select from these components depending on his current requirements. The designer selects a software module or component based on meeting his functional and interface requirements. If a component does not already exist which meets the requirements, the designer must develop a new component. The designer can create new components from scratch or he can create new components by combining or modifying existing components.

The techniques for transformational programming, software synthesis, and program generation may differ on: what knowledge is contained in the knowledge base and what it consists of, how it is represented, how new knowledge is added, and what functional transformations exist that can be applied to the knowledge and how these transformations operate. However, there are more similarities among these paradigms than differences.

With any of these techniques, the software evolves through transformations and refinements performed on the original specification. The designer performs these refinements in such a way as to ensure consistency, correctness, and completeness with respect to the original specification (41). Figure 2.6 depicts the Transformational Implementation (TI) model first proposed by Balzer in (9) and described by Fickas in (41).

The designer starts with a formal specification, and then applies a transformation process to produce a new refined specification. The transformation process may be repeated until a specification version is produced that meets the implementation conditions. The TI model can be applied to a formalized automated system, generating an automated transformational programming environment (41).

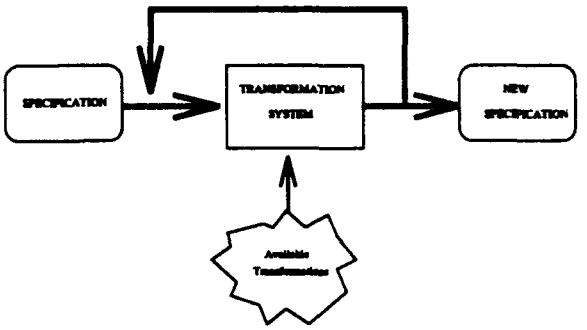

Figure 2.6 Transformational Implementation Model

The TI model is consistent with the general transformational programming paradigm described in Chapter 1 (shown in Figure 1.1). Balzer updated his TI model in (11) for its implementation in an automated environment. Two specific phases of transformation identified for this automated environment were *formalization* and *optimization* (11, 13). Both Balzer and Fickas proposed methods of automating the transformational model through addition of automated support tools and software libraries contained in a knowledge base.

The supporting tools can be complemented with knowledge-based assistance, providing the designer additional interactive guidance. The guidance may be based upon transformation, synthesis, and inference techniques. Current research includes varying degrees of automatic and interactive program development (11, 41, 93, 117). For example, the transformational programming paradigm, as identified in (2) and shown in Figure 1.1, includes the possible use of an expert system.

2.4.2 Existing Automation Systems and Techniques. The following sections describe various automation systems and design techniques whose characteristics related to this research.

2.4.2.1 Hierarchical Software System. Don Batory and Sean O'Malley developed a model of hierarchical software design based on interchangeable software components (14:2). They defined *components* as their fundamental units (software). A *realm* consists of those components that share common interfaces. The engineer can combine components, based on composition rules, creating new components and subsystems; establishing a design hierarchy.

The overall design process described in (14) is incorporated into the methodology of the proposed parallel software development environment described in Chapter 3. An analogy between a realm and a domain can be drawn, and a domain model would then consist of the realm and its associated rules of composition.

Batory and O'Malley equate their design hierarchy scheme to that of a grammar, where the productions define the rules of composition, and the terminals represent the components (14:4-5). Their method of description shares similar concepts with the CSP language model (57). The underlying concept of all these methodologies is an operational semantic based on a transition system.

2.4.2.2 KIDS. The Kestrel Interactive Development System (KIDS) is an example of an automated development tool that applies the concept of software synthesis. KIDS provides a set of tools to transform specifications into code. An extensive knowledge base for establishing domain, problem, and algorithmic theories distinguishes this system from other research efforts. The system utilizes the knowledge base for inferencing techniques to synthesize program generation. KIDS includes tactics designed to preserve correctness of the program at each step of the transformation process. The first step is to develop a domain theory, which must be preserved throughout the process to ensure the final product meets the specification and does not contradict the domain knowledge (116:1025). KIDS is built upon the REFINE programming environment (103).

2.4.2.3 Domain-Specific Automatic Programming. David Barstow defines automatic programming as a system that "allows a naive user, one who does not necessarily understand programming, to describe a problem using the familiar concepts of the domain" (13). Researchers have implemented automatic programming systems that concentrate on either the formalization of the description, or the optimization of that description into an implementation version (10, 13, 41). The differing objectives of these two stages of development have led to a divergence of systems.

Each system currently automates a different portion of the whole process and each takes a different approach to the transformation process based on its specific objectives.

Barstow makes a distinction between three different views of problem description: informal specification, formal specification, and algorithm specification. The transformation process first evolves the specification from the informal to the formal. The designer then decomposes the formal specification into the applicable data and control structures for algorithm description. The algorithm description evolves from the abstract to the concrete through a series of optimizations.

Throughout the entire design process, domain knowledge can be applied for the formalization and optimizations. The domain objects may be formally defined through mathematical structures, allowing the user to apply these constructs in the initial description. In addition, domain knowledge can be applied to define the rules of transformation applied to optimize the description for a specific implementation. Barstow's research used the GIST specification language (9, 13, 40).

Barstow stated that the two fundamental problems facing the design of an automation system is: "How should the domain knowledge be represented for use by an automatic programming system?" and "How do domain knowledge and programming knowledge interact during the programming process?" (13). These questions require answers for the development and implementation of any automated software development environment.

Barstow's research shares several common features with current research efforts of the AFIT Knowledge-Based Software Engineering group (6, 73, 102).

2.4.2.4 ARCHITECT and Other AFIT Research Efforts. The Knowledge-Based Software Engineering Group at AFIT applies transformational programming techniques and domain analysis concepts in its research. Several students worked on the creation of the model called *ARCHITECT* (6, 102). ARCHITECT uses the REFINE programming environment as the underlying design platform. REFINE provides a formal specification environment for the design of software systems. The software architecture underlying their model is the OCU architecture described in

Section 2.2.5. The specific domains used to demonstrate their research include logic circuits and digital signal processing.

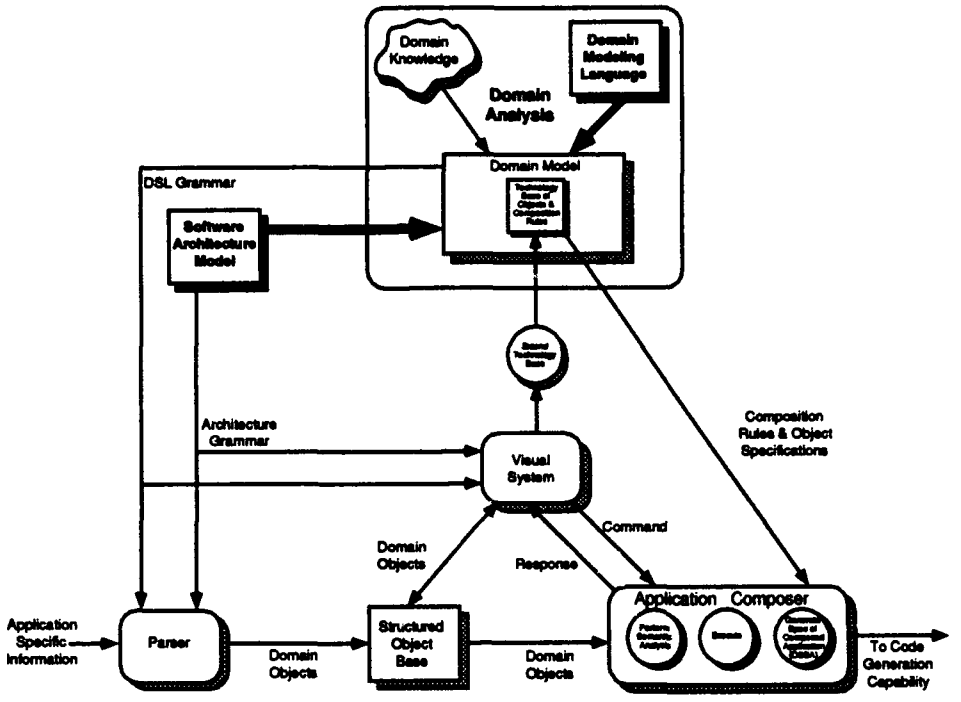

Figure 2.7 ARCHITECT

The ARCHITECT environment uses concepts of domain-specific software design and software architecture concepts within the development process. See Figure 2.7 for an overview of ARCHITECT. Their process begins with the definition of a formal domain model. The software architecture model and domain model define a specific structure for the knowledge base. The knowledge is used in conjunction with a parser to formulate a specific domain language. Using application specific information combined with this domain-specific terminology, a formalized specification is generated. Through an iterative refinement process, the formal specification would be transformed into an automated solution of the application problem realized in executable code.

The ARCHITECT system design has been expanded and generalized to the Domain-Oriented Application Composition Environment (Figure 2.4.2.4), adding a more sophisticated visual interface (31), composition system (53), and knowledge base structure (technology base) (22). Additional

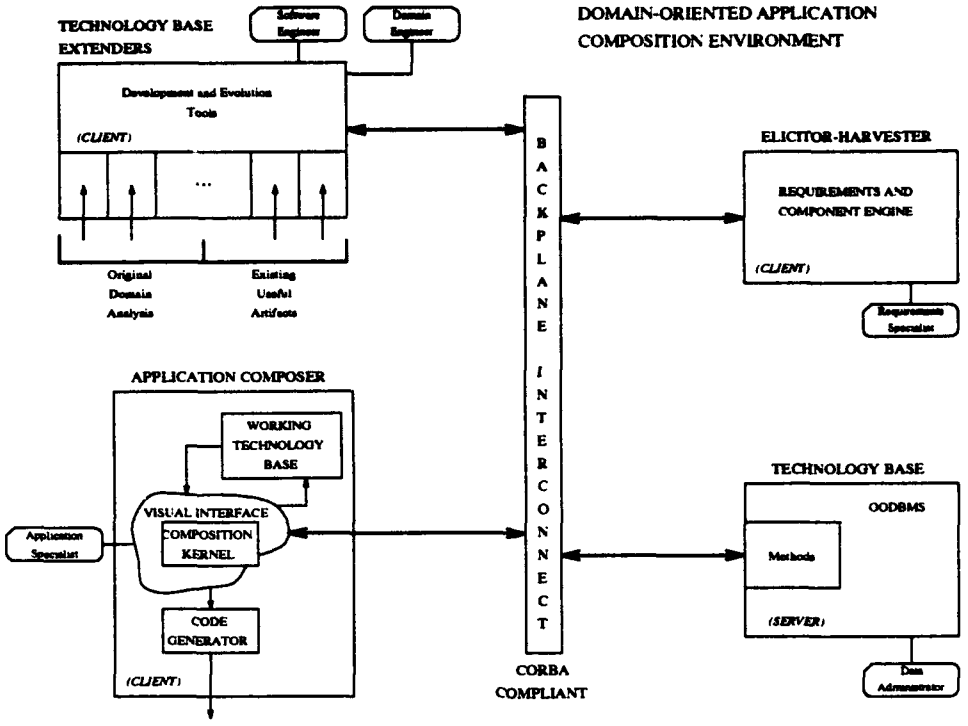

Figure 2.8 Domain-Oriented Application Composition Environment

research efforts evaluated an extension to the technology base by investigating alternative environments and their importation into the AFIT system (109, 123). Future research efforts plan to investigate the addition of a requirements Elicitor-Harvester to assist the designer in defining and formalizing the application requirements.

The platform envisioned in this thesis incorporates much of the knowledge and experience gained through AFIT's prior efforts. However, the KBSE group's research thus far has focused on sequential computation. Our research will apply these concepts to the parallel processing environment.

Lamont and Luginbuhl modified the ARCHITECT model to create an alternative parallel model. Their initial model proposed the combination of REFINE and UNITY language features (see Figure 2.9). Their research effort will investigate the application of this model on NP-complete optimization problems. This thesis extends their initial efforts as described in (72). Future efforts

will explore a mapping between existing UNITY specifications of NP-complete problems and an extended REFINE environment.

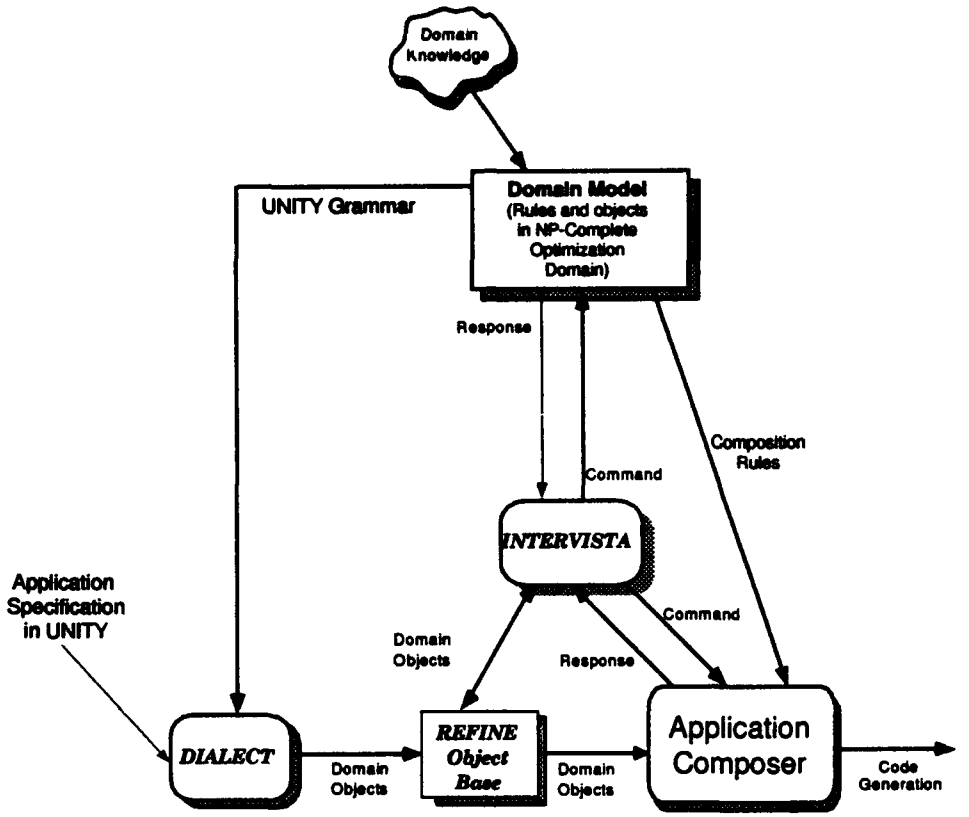

Figure 2.9 Domain-Specific Parallel Software Environment with REFINE and UNITY

2.4.2.5 Task Management Tools. Task management methods use graphical modeling techniques and design heuristic algorithms to attempt to efficiently and effectively map an application to a specific implementation architecture. The application is captured in a task system software architecture composed of a problem graph and an architecture graph. A designer generates an execution model of the system by creating a mapping between the two graphical models. The current capabilities of this modeling technique has been demonstrated effective only for static applications, where a trace of the execution behavior is known *a priori*. The dynamic behavior of the more sophisticated parallel applications requires further research.

A detailed discussion of the task management issues is provided in Chapter 4. Appendix D expounds on the issues of task management in the specific areas of scheduling and load balancing. Included in Appendix E is a description of specific algorithms and tools for task management and software design. Of specific interest to this research were the sophisticated environments provided by TASKGRAPHER (80), PARSA (111), and PYRROS (51).

2.5 Chapter Summary

Thus far, we have examined the methods designers use to model the real world in creating software solutions to the application problems. Through abstractions and mappings, designers capture the properties and behaviors of the desired system within specific models. The models provide software designers a better understanding of the properties and behaviors of the problem being modeled. By using models based on formal properties of mathematics, they can formalize aspects of the software design process. The formal models allow for logical analysis and proof of design methods.

The use of models within software engineering is fundamental to the development process. However, researchers advocate various modeling techniques (55, 77, 36, 78). The choice of models impacts the organization of the domain knowledge and the interaction of the domain objects.

Through the incorporation of domain modeling and software architecture concepts, the designer defines a specific method of knowledge representation. The structuring provided formalizes the development process and allows the the use of the inherent properties and knowledge encoded in the models (12, 55, 76, 94, 117, 124). The importance of the designer's selection between alternatives for knowledge representation was identified by Barstow (13) and determines how the program and domain knowledge interact within the software development process.

The formal specification environments provide the ability to describe application problems using languages having formal semantics. The formal semantics allow for domains where mean-

ing can be defined for properties of consistency and correctness (55). The formal languages and environments will be evaluated further in Chapter 5.

The transformation process allows for a consistent and correct mapping between one representation of a problem and another. The transformation process can be formally defined to allow automation of the program refinement. Several existing software development methods apply the transformational programming paradigm for software development.

Our design combines concepts of software engineering modeling, domain-oriented design, software architecture modeling, transformational programming, and formal specification design methods for application to parallel software development. The next chapter will provide an overview of our research approach and a description of the conceptual platform envisioned for parallel software design.

III. Methodology

3.1 Introduction

This chapter provides an overview of the conceptual software design environment and a general description of the objectives for the related research areas. The conceptual environment is described in Section 3.2. The description incorporates a more detailed description of the application of the domain-oriented design and software architecture modeling concepts. The roles of the various individuals involved in the process are also characterized.

The environment's description is followed by an overview of the research issues and objectives in Section 3.3. This section provides a discussion of the issues in defining the theoretical environment. Although the software development process is generally characterized in the literature, the implications of the incorporation of concurrency have not been addressed, this is the goal of our research. This section describes the research areas we evaluate and the objectives for these evaluations.

Before describing the conceptual environment, it is important to emphasize the executable nature of the design specification. The validation and verification of the specified behavior requires that the design specification be executable (122). The operational semantics of the language define the problem in relation to an environment and provides the execution capability.

As described in Chapter 1, the general transformational programming paradigm (shown again in Figure 3.1) uses a transformation process to generate an implementation. Two specific phases of the transformation process can be identified: *formalization* and *optimization* (11, 13). The specifics of these development phases will be defined in further detail later in this document.

The transformation process starts by generating a formalized specification of the application. By using an operational semantic, we can ensure that the formalized specification is executable. Thus it may be viewed as an operational specification (2). However, one must recognize that at this point in the development process the specification has not been optimized for implementation.

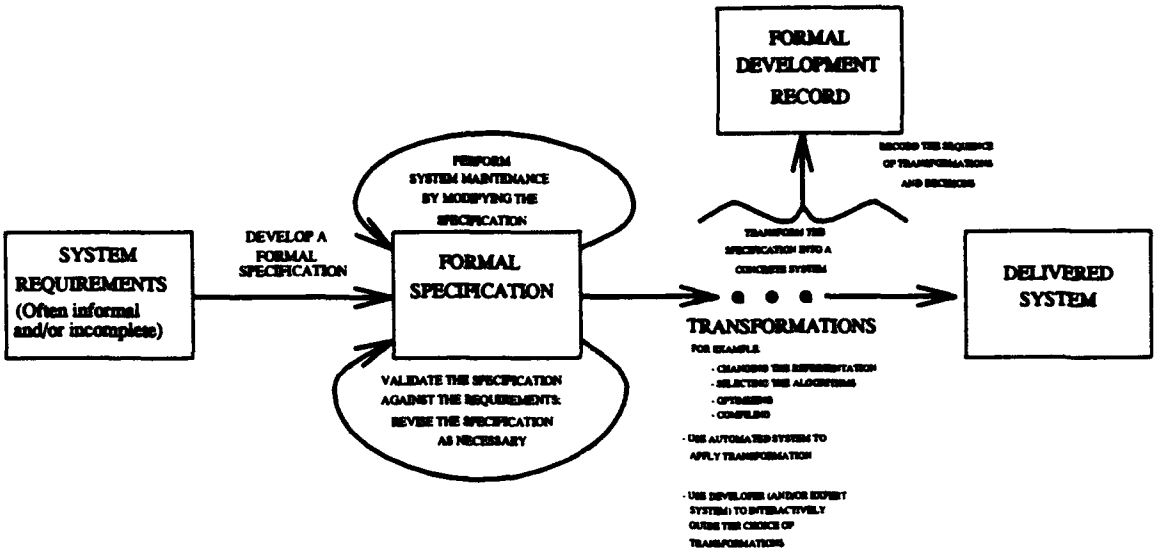

Figure 3.1 The Transformational Programming Paradigm

Although the specification is executable, the focus of the execution validation is on consistency, correctness, and completeness, and not performance efficiency.

In (120), Swartout and Balzer refer to their categorization of the software development process as *specification* and *implementation*. Clearly, there is a strong relationship between this classification and the classification defined by Barstow in (13) of *formalization* and *optimization* (13). Both refer to the intertwining of the two phase of the software development process for the realization of better products.

Swartout and Balzer point out that "a specification is itself an implementation of a higher-level specification. Each step in the development process should be a *valid* realization of the specification" (120). An implementation's validity is relative to its inclusion of a proper subset of those behaviors defined by the specification. As one progresses to a more detailed specification, the designer becomes more concerned with efficiency.

The defining of the domain grammar within the domain analysis process must account for the difficult decisions of the implementation versus problem view in defining its operational semantics. The operational semantics will define the aspects of behavior that will be validated within the re-

finement process. It is critical to the formal verification and validation process that the execution of the specification maintain the consistency and correctness of the specification description; however, the structuring of this knowledge must also support the later optimizations for efficiency.

3.2 The Conceptual Platform

The conceptual platform for this research is depicted again in Figure 3.2 . A knowledge base is at the heart of this conceptual process, and thus it is considered a knowledge-based software engineering approach. The platform itself cannot be used until the knowledge base has been loaded with the domain-specific and software design knowledge. The knowledge base is organized into several subdivisions based on their specific contribution to the overall process.

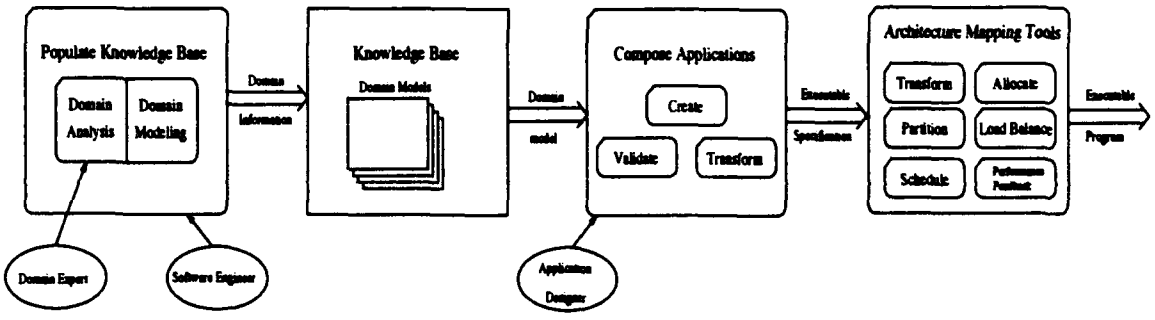

Figure 3.2 Domain-Oriented Composition System for Parallel Software Development

The problem is formalized through the defining of the domain grammar (through domain analysis) and the formal composition of the domain entities into a specific software architecture description. The system allows the iterative refinement of this process (through additional compositions and transformations) until the validity of the desired behavior(s) is realized. The designer can then apply the support tools to optimize the implementation. The designer transforms and refines the problem to obtain greater efficiency, while continually validating the behavior(s). The refinements add greater details for the data and control structures. The designer can apply additional tools for the allocation, partitioning, scheduling, and load balancing of the problem implementation. The details of the overall process are described further below.

Personnel with various specializations interact within the development process. A detailed description of their titles and various roles is described in Figure 3.3. The titles are used to identify the various specializations required to perform the activity and are not precluded from being the same individual (6).

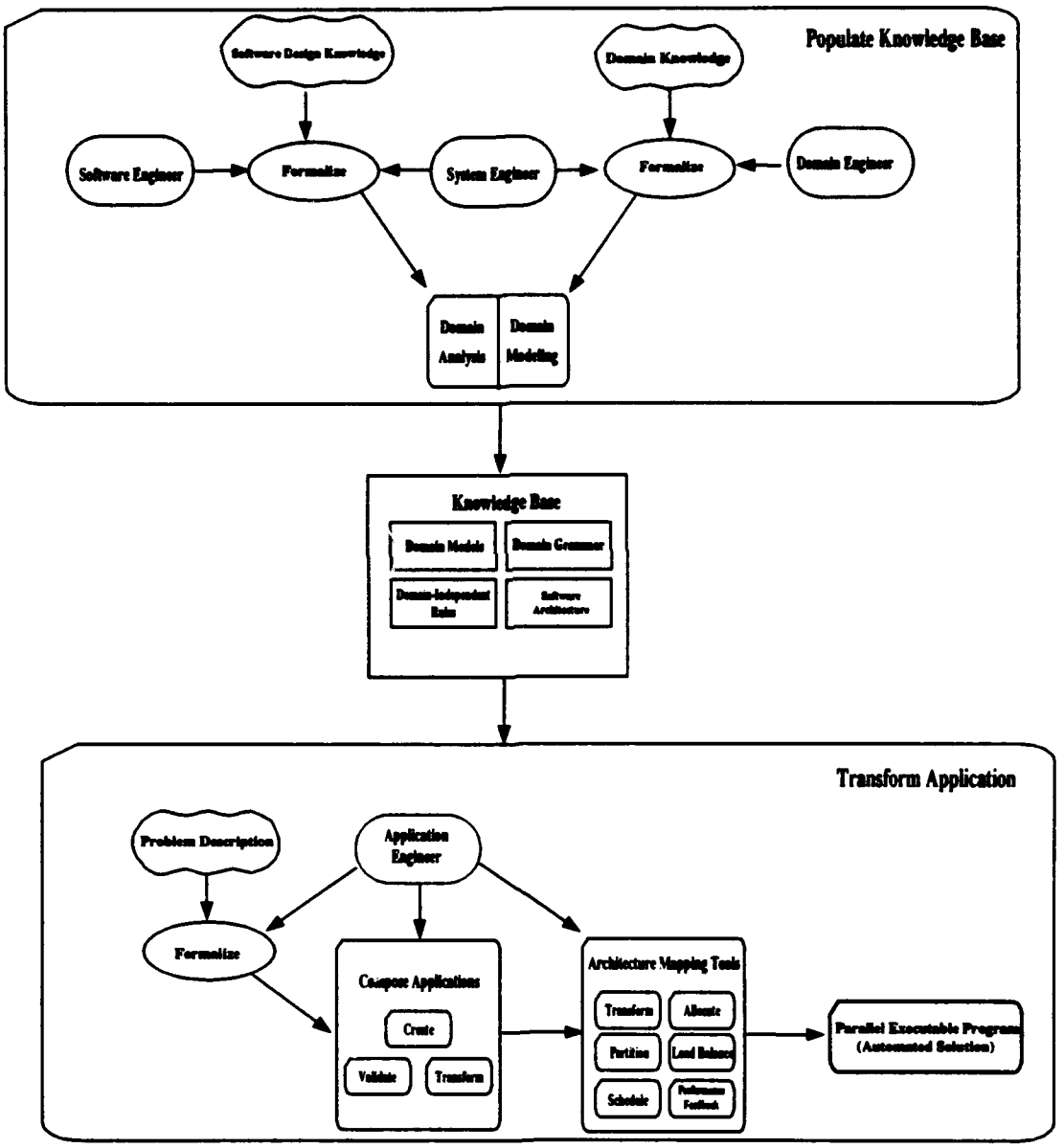

Figure 3.3 The Process and the People

1. System Engineer: Works with the domain engineer to partition the domain problem into system requirements. Partitions the tasks to software or hardware requirements (8). Identifies the impact of system requirements on domain knowledge representation and software design knowledge representation.
2. Domain Engineer: Possesses detailed knowledge about the domain and gathers all the information pertinent to solving problems in that domain (71:4). Models the real-world entities required to satisfy the policy, strategy, and use of an application as defined by the system analyst. Determines how, if possible, these entities can be modeled within the constraints specified by the software engineer (6).
3. Software Engineer: Designs new software systems in the domain (71:4). Responsible for defining a formalized structure for the domain knowledge and providing the translation from the domain-specific terms to executable software (6). Encodes software design knowledge and design techniques into specific composition rules, modeling techniques, and transformation methods.
4. Application Engineer: Uses systems in creation of computational model. Familiar with the overall domain and understands what the new application must do to meet the requirements (*a sophisticated user*). Provides the application-specific information needed to specify an application (6). Composes an application using domain knowledge and software architecture concepts. Validates the executable specification's behavior based on desired behavior. Performs multiple transformations to refine and optimize the specification, re-validating as required. Maps final specification to implementation architecture (mapping may require a recomposition of problem based on a selected architecture).

A *domain engineer* uses domain analysis to create a domain model. At a minimum, the domain model includes the various domain objects; their operations, relationships, rules, and functions for application composition. The domain knowledge for theorems and axioms of the domain is also included in the domain model. The domain model includes domain-specific compositional rules and composition operators for the formal composition of the application specification. The application specification contains an abstraction of the software architecture.

Working with the domain engineer, a *software engineer* captures the specifics of the domain model using the underlying wide-spectrum language. The software engineer uses the primitive types and functions to create a hierarchical description of the domain or domain taxonomy. The hierarchy represents a taxonomic hierarchy of knowledge identified by Barstow (12) and may be parsed to an abstract syntax tree representation based on the domain objects. The hierarchy formalizes the relationships between the various objects and categorizes them based on domain-specific terminology.

The software engineer, with support of various object base constructs and tools, uses the constructs represented in this abstract syntax tree as the foundation of the domain-specific grammar. Since the syntax tree is rooted in the low-level mathematical constructs of the wide-spectrum language, this process creates a formalized grammar and an operational semantic for the domain-specific information. Each of the domain-specific objects can have additional constraints and attributes, creating a schema for each term. Object operations are associated through their functional (domain/range) specification.

The software engineer also imposes specific composition and transformation rules based on a specific software architecture model description. The model can abstract specific properties of the application composition and can also represent this as implicit knowledge. Rule-based techniques can be applied to synthesize the implicit knowledge contained within these model descriptions.

The knowledge base includes additional non-domain-specific functions, rules, and transformations, which represent problem strategies (such as branch and bound, divide and conquer, and breadth-first). The traditional concepts of design captured by this codified software design knowledge facilitate the transformation to an acceptable solution. The software design and domain-specific knowledge interact, along with the user, to create an automated assistance environment.

The knowledge base is populated with various pieces of knowledge to define the domain grammar, compose a specific application problem's software architecture, and guide the transformation to an implementation version of the application.

Once the domain-specific model is loaded into the knowledge base, an *application engineer* may use the application composer to generate a high-level specification. Because the specification is generated from the constructs of the domain-specific language, the initial specification provides a high level of understandability and readability by using terminology familiar to the experts within the domain. The price paid for this is the difficult task of domain analysis for formalizing the domain knowledge.

A parser can also be used to generate an abstract syntax tree of the domain-specific objects included in the software architecture. The parser and composer use the information within the knowledge base for this domain-specific process. The domain-specific rules, guidelines, theorems, and axioms constrain the composition of the domain objects, assisting the application specification process.

The extensive tools of this parallel software development environment may then be brought to bear to perform domain-specific syntax and semantics checks. These evaluations provide additional checks for validity, consistency, and correctness. Since a language with a formal semantic is used, the specification can be evaluated for consistency and correctness. Since the specification represents a formalized version of a software architecture with an operational semantic, the specification may be executed for validation of desired behavior.

The composition of objects includes the ability to specify concurrency within the software architecture created. Based on the behavioral feedback from this prototype, the composer performs refinements to the specification until acceptable behavior(s) is achieved. Acceptable behavior(s) is determined by the specification's execution, obtaining the desired behavior(s) required by the high-level description, validating the specification. It is important to note the possible plural nature of behavior. Because concurrency is a fundamental part of our specification, the design must allow for nondeterminacy in behavior, in other words, a set of allowable behaviors. Therefore, this process can also be used to validate an acceptable and effective level of concurrency. For example, the application engineer can specify activities to occur concurrently and determine if the desired (and therefore correct) behavior is maintained.

At this point, the application engineer has not optimized this initial executable model for efficiency. The application specification has been generated to focus on the completeness and correctness of the desired behavior(s) of the application — a formalization of the problem. The next stage is to optimize this specification in preparation for implementation.

Once the application engineer has validated the specification, he may additionally apply a number of other software transformation or synthesis techniques to transform and refine the high-level syntactical version of the specification to lower-level versions. The purpose of these refinements is to provide a more concrete description of the problem. These refinements are at the bridging point between formalization and optimization.

The focus of the optimizations is to increase execution efficiency and eliminate any redundancy. An example of an efficiency transformation would be the selection of an appropriate data structure. As shown in Figure 3.4, different data structures have difference performance parameters. An example of redundancy would be a specific constraint that has been specified repeatedly or specified through both explicit constraints and implicit constraints where either one or the other would be sufficient.

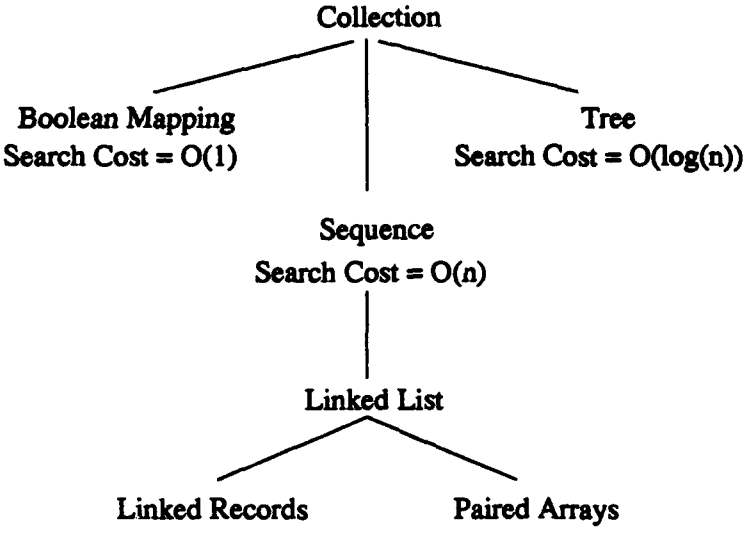

Figure 3.4 A Taxonomic Representation of Alternative Data Structures

The application engineer may apply additional transformations to tailor the specification to a specific hardware platform and execution language. He has the capability to apply synthesis techniques, to use heuristic decomposition, and to use scheduling methods on the problem. These techniques assist in the efficient multiprocessor mappings. The architectural mappings must provide some means to account for any topological or cardinality variations between the hardware

architectural and the problem (software architectural) models. A detailed description of the difficulty that these problems impose on the application is covered in Chapter 4 and Appendices D and E.

In summary, the overall transformational development process operates through an interactive and automated system. The system allows the refinement and tailoring of the specification to optimize execution of the application for a specific parallel system. With a software development process based on formally defined composition and transformations, the user ensures the refinement process maintains the correctness and consistency with respect to the original specification.

The addition of knowledge-based automation tools with domain-specific information provides for a domain-oriented design process. The user may specify the application problem at a higher level of abstraction with terms having a direct interpretation in the domain. Domain-specific knowledge (contained in the knowledge base) may guide the transformation process, easing the search for an acceptable solution.

The software development process envisioned in this thesis provides a means to both formalize and optimize an application specification. The formalization occurs through the formalizing of the domain model and the formalized composition of a software architecture. The domain analysis provides formalized domain objects, operations, rules, transformations, and constraints. The composition system provides a means for assembling a software architecture that additionally maintains consistency with respect to the domain model. The composed model can be simulated in an execution environment to verify the behavior(s) and the model's consistency and correctness to the desired or acceptable behavior(s). Once validated, the designer can optimize the software architecture model through the various available transformations and mappings. Optimizations are further realized interactive tools to map the application to a specific execution language and parallel machine.

3.3 Research Issues and Objectives

This section provides an overview of the specific issues and objectives of various aspects of our research. The information described thus far provides a general background towards a greater understanding of the problem of developing the described conceptual environment. The information provided has begun to highlight specific issues that need to be addressed. Later chapters describe and analyze various existing tools and methodologies for application to the conceptual environment described. This section lays a framework for this analysis.

As described above, the characterization of the issues faced by our research can be divided into the formalization and optimization phases. The issues of formalization focus on understanding the domain analysis process, software architecture modeling, formal specification process, and automated environment for assisting in the formalization process. The issues of optimization require an understanding of the transformation programming process and parallel software implementation issues. Researchers are just beginning to understand the significant issues of task management as applied to parallel applications (26, 88). In addition, one must understand the impact of concurrency on these concepts. A complete environment must encompass all these strategies.

The formalization process is impacted by the level of abstraction and formality in the initial specification version. As described in Chapter 1, the designer must make a trade-off between understandability and the level of formality (preciseness in mathematical definition). The characteristics of this initial specification affect the resulting transformation phase (2, 10, 94).

For example, the specification may be specified using domain-specific objects, relationships, and operations. The use of the domain-specific terminology enhances the readability and clarity to the user, but may lack efficiency in the operational semantic. Thus, the specification would require a complex transformation process for mapping the domain-terminology to a lower-level, more efficient, concrete implementation. The alternative of specifying the application at a lower

level would alleviate the amount of transformations, but it would require a much more complex process for generating the initial specification (2, 13, 98).

Our approach provides a paradigm closer to the former of these alternatives. A complex knowledge base structure is used to formalize the domain knowledge for use in problem specification. The initial problem description is formalized at a higher level of abstraction, providing a clear and understandable description of the problem. The complexity is therefore in the analysis and tools of the development process.

The optimization process focuses on making the formal specification (or operational specification) more efficient. The evaluation of this specification is not concerned with the readability of the specification. The efficiency is a measure of the performance (or expected performance) in the operational environment. The designer may apply transformations to select from alternative data structures or control structures (algorithms).

The differing objectives of the formalization and optimization phases has led to a diversity of automated systems, specification environments, and specification languages (10, 13. 12, 93). The analysis performed for this research seeks to find a relationship between the different transformational processes and apply a unified concept to an environment for parallel software development.

The parallel software design requirements add a new dimension to the problem; the mappings, allocations, partitions, schedules, and load balance of the operational specification for a specific parallel architecture add to the combinatoric complexity of the optimization process. These task management techniques (further defined in Chapter 4) attempt to optimize the design for performance. The optimization process in this environment raises many performance questions. For example, the specifier needs to answer the following: Does the application require the efficient optimization of the problem for implementation on any platform (portability) or for a specific platform? — a question impacting problem definition and scalability. Traditionally, these non-functional requirements were not considered until the later stages of the process.

Adding parallel issues to the formalization process requires the identification of the inherent concurrency within the application; however, merely identifying and specifying concurrency is not sufficient. The consistency and correctness of the parallel implementation requires the architecture mapping(s) to maintain specific constraints of the problem. These issues translate into specific criteria for the language's operational semantics and the control structure of the implementation. The existing formal specification environments and formal languages use alternative methods for overcoming these problems; however, these environments have several similar conceptual views of concurrency representation.

3.4 Chapter Summary

Before concluding this chapter, we summarize what we consider to be the major issues that need to be addressed to make the conceptual parallel software development environment a reality.

3.4.1 Automation Systems for Transformational Programming. The discussion and analysis of the modeling techniques were provided as part of the literature review in Chapter 2. In addition, that chapter provided a description of some of the current research on knowledge-based software engineering and software automation techniques. The current research demonstrates that a formal, transformational programming environment is possible. The commercial products REFINE and KIDS are examples. But are these methods applicable to more than just pedagogical examples? And can they be extended to include concurrency or parallelism?

3.4.2 Parallel Design. The design of a formal environment for the development of parallel software requires an understanding of the pertinent issues of concurrent design. Specifically, we need to evaluate parallel hardware architecture issues (Appendix C), parallel software design issues (Chapter 4), parallel performance issues (Appendix B) and task management concepts (Appendices D and E, and Chapter 4).

3.4.3 Formal Languages and Formal Specification Environments. The existing languages and environments provide a great deal of insight into the required features for a parallel specification language. However, there is a disparity between the different environments due to their focus on either formal method or optimization techniques. We evaluate the benefits of the properties of each of the formal environments for incorporation into a conceptual design for the complete parallel software development environment proposed in this chapter. Chapter 5 lists a description and analysis for several of these environments.

3.4.4 Conclusions. This chapter described a conceptual platform for parallel software design. The platform combines existing formal and informal techniques of software specification, modeling, and design. The chapters following describe the analysis required to gain a better understanding of the design of the desired platform. The existing formal specification environments and optimization tools demonstrate that the technology to create the envisioned parallel software development environment (as proposed in this chapter) is currently available. This thesis describes a conceptual design for integrating this technology through formal parallel language constructs and the proposed environment.

IV. Designing for Concurrency: Issues of Parallel Program Design

4.1 Introduction

The concept of sequence is fundamental to human thinking and natural law. We base our daily schedule on a sequence of activities in time. We communicate with one another through a sequence of words, either spoken or written. Everywhere in human life and nature there is an inherent sequence. But throughout our world there is also a great deal of concurrency or parallelism. Within our universe sequences are occurring simultaneously, adding to the complexity of our space/time understanding. As part of an organization, individuals cooperate and coordinate activities toward a common goal. In nature, electrons orbit about the atoms of electrical and magnetic fields simultaneously. Thus, to clearly understand and model the world around us, we must understand and characterize both the sequential and concurrent levels of observation (abstraction).

Computer science has gradually matured to include concurrent concepts within the specification and execution of programs. In this chapter, the impact of concurrent concepts on various aspects of parallel computing design is described. The information presented in this chapter relates specifically to the optimization and implementation issues of the parallel software development environment described in Chapter 3.

Section 4.2 describes the hierarchy of abstraction and its relationship to transformational programming. The concepts are related to the problems of the semantic gap. Sections of this chapter address each level of the abstraction hierarchy.

Section 4.3 describes the relationship between the parallel hardware architecture design and parallel software design and the issues faced by parallel software designers. It describes the relationship between an application's communication and synchronization dependencies and their implementation through a mapping (transformation) to the parallel system.

Section 4.4 describes parallel algorithm design, including problem dependencies, the use of explicit parallel language constructs, parallel compilers, and the relationship between parallel al-

gorithms and concurrent operating systems. It addresses the problems that limit parallel program performance and provides techniques for solving them.

Section 4.5 describes parallel design from the task management viewpoint, describing concurrent processes as interacting tasks. Through task management, one generates a control structure to effectively map the tasks to an underlying hardware architecture and manages the task structure for efficient performance.

Finally Section 4.6 summarizes the chapter and relates traditional concurrent design to the proposed parallel software development environment. Readers are assumed to have an introductory background in software design, algorithm design, parallel performance terminology, and parallel programming. As stated previously, the appendices of this thesis provide background into some of these areas. More detailed information on parallel programming and architectures can be found in (3, 38, 30, 32, 45, 79, 101, 118).

4.2 Characterizing the Processor View

Chapter 1 described the basic issues associated with the problem of the semantic gap for software development. This section analyzes the problem in greater detail from the *processor view*, looking at the specific issues of the hardware architectures in the parallel environment from a top-down perspective with respect to the hierarchy of abstraction. Understanding the levels of abstraction is important toward understanding and solving the problems associated with the semantic gap.

The basic process of computation for a given processor is the same no matter which architecture is involved. The processor takes data in, processes it, and produces output. As one goes from the high-level software design to the low-level processor view, the level of abstraction changes, thus changing the designer's observation of the problem. The designer may observe little apparent structure in low-level machine code generated by a compiler. The compiler may have restructured the

instruction ordering to optimize the execution for efficiency. A direct relationship to the high-level design description may be difficult for the designer to perceive. However, if the problem encoding and the compiler mapping are both correct, these different views of abstraction should have the same semantic (meaning to the designer).

We can characterize the levels of abstraction in a top-down analysis as shown in Figure 4.1.

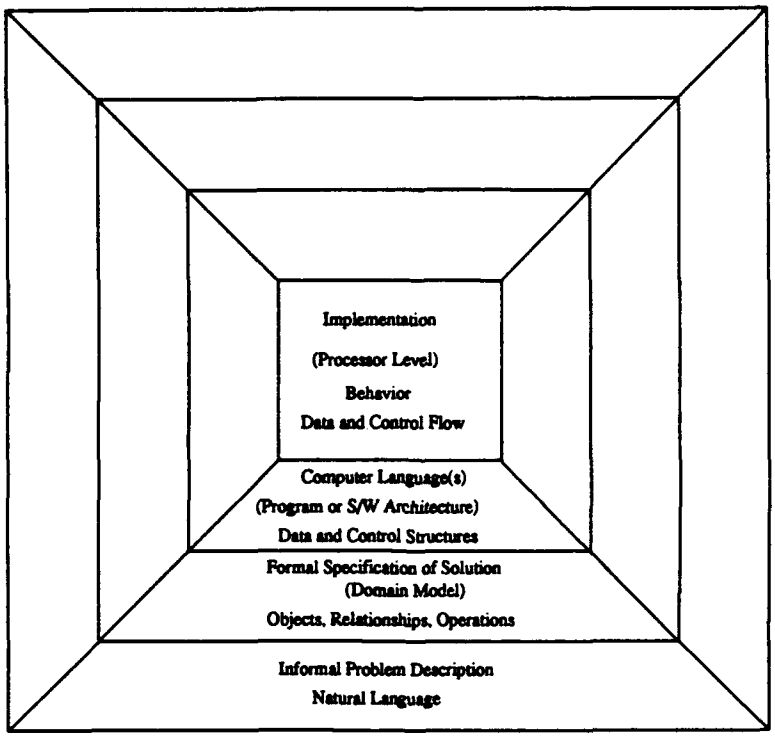

Figure 4.1 Levels of Abstraction

- The application is characterized by informal ideas or concepts. A user generates these concepts and ideas based on a problem he wants to automate.The designer creates a formal specification (solution) based upon an analysis performed on the informal description of the application problem.
- A formal specification of the problem solution is generated through design analysis. The formal specification characterizes the problem solution based on the specific objects, relationships, and operations of the applications (which may be represented using abstract data

types). These concepts may be common for a specific application class and therefore may be generalized for a problem domain. The program itself is an encoded version of the actual application problem solution.

- The designer structures the computer language view of the problem using data structures (abstract data types) and control structures (algorithms). Designers abstract this problem view through use of a higher-level computer language, where a compiler applies a mapping from a computer language to the specific implementation machine code. The compiler automates the process using language-specific mappings to generate machine-specific code, and may apply environment-specific (for a specific operating system) transformations.
- The application is defined by the machine code instructions. The program's implementation generates specific machine behavior(s). The machine behaviors are characterized by specific control signals (the data flow and the control flow).

The overall process of design evolves through applying mappings between these abstract views of the problem solution. The process may include partial mappings or informal mappings; however, these mappings must maintain a consistent form to preserve the correctness between the user's desires and the machine's behavior. The transformational model described in Chapter 3 requires the mappings to be formalized to maintain consistency and correctness. In the following section, the problem of software design is characterized by a relationship to the machine abstractions for data and control flow.

4.3 The Processor Level

The designer's view of the system as a virtual machine can be created by abstracting the control structure to obtain the program data. The data are assumed available after some given time (latency) and specifically *how* the required information is obtained is abstracted. Using this abstract or virtual machine model, the differences between architectures are seen as variations in

latency times for accessing and receiving the needed data. The designer can use this concept to create a functional view of the system. This virtual processing view simplifies the model for the designer.

Unfortunately, in order to realize the efficient speed-up in implementation, we cannot abstract completely this control structure from the design issues. The control structure dictates the control flow on the parallel machine. The synchronization between processors to achieve the desired processor cooperation requires the use of a control structure for passing data.

The control structure is also important for maintaining the consistency and correctness of the described specification (solution). The specification describes specific concurrent behaviors that must have a specific operational semantic for the machine implementation. By *control structure*, we mean that portion of the system that manages the data and control flow of the machine's execution (which may be represented as a specific algorithm). The overall machine control structure consists of many cooperating and distributed structures.

Note that both hardware and software are used in the overall machine's logical control structure implementation and there may be a hierarchy of execution. For example, a control structure in the form of an operating system is required for ensuring the program's consistency by maintaining data coherency. A control structure is also required to regulate the appropriate data and control flow for the program's efficient execution. In addition, memory accesses are also performed through a specific memory management control structure. Throughout computer software and hardware design, specific features that manage the interaction of data are referred to as control structures or algorithms.

Of specific interest to the parallel software designer are the latency time's impact on performance and correctness of behavior. For example, the correct execution of distributed tasks may require a specific order in execution. The designer attempts to preserve the correctness through specific task synchronizations. The sharing of data between processors requires a specific data

access time. Because the latency time has such a dramatic impact on all areas of performance, it has become one of the primary issues of hardware and software design (32). The designer uses various hardware and software design methods to attempt to reduce the effective latency time for a processor. For example, electrical engineers work to create hardware designs to reduce memory access time and to speed up interprocessor communication networks, while software engineers generate efficient scheduling algorithms to map the problem to the processors to reduce the need for interprocessor communication.

The problem's required communication and synchronization are dependent upon its decomposition and partitioning, inherent algorithm control structures, the processes' scheduling, and mapping to a specific parallel system. The relationship of these problems to the hardware architecture may require machine-dependent design solutions. For efficient design, the variations in hardware and software designs to overcome the latency cost have led to variations in implementations and problem specification.

Different algorithms have been shown to have different levels of efficiency and effectiveness for different system architectures. Thus to achieve the desired efficiency, the designer may be required to tailor a problem solution identification and algorithm and data structure design for the specific parallel architecture. For example, distributed systems require the use of explicit communication protocols for interprocessor communication and data coherency, whereas shared memory machines merely access a protected shared variable (79, 80).

Various methods for overcoming the limitation of the communication latency include improved task assignment and scheduling algorithms, faster and more efficient communication interconnection networks, use of communication coprocessors, multitasking processors, improved memory design (memory hierarchies), and more efficient memory management control structures. These features add to the alternatives that must be considered in the design process and may impact the problem definition and identification process. For example, the addition of multitasking within processors

provides flexibility for potential groupings of tasks and mappings. By having the ability to context switch between tasks, a processor may be able to reduce execution time by hiding the effective communication latency (47). The decomposition of the application tasks and their allocation to processors must account for the possible use of multitasking, otherwise this feature cannot be effectively used. With respect to the control structure required for process synchronization, we note that the multicomputer and multiprocessor differ slightly. In a shared memory structure of a multiprocessor, barriers are used to control access to shared variables. Designs use shared variables to synchronize the execution; control can be implicit (contained in memory management) or explicit (requiring a specific set of instructions to generate an internal control structure). In the distributed environment of multicomputers, explicit receive and send instructions are used to synchronize processes. The designer may create specific rendezvous points and barriers to synchronize the overall program execution.

The processor's control flow (defined for a specific implementation) may also be impacted by data and control dependencies between tasks. The identification and decomposition of the problem must consider these dependencies in the design of the problem's control structure (algorithm design). The tasks may either require a strict deterministic execution or allow for nondeterminacy. Researchers have found that the specification allowing for nondeterminacy in action provides the greatest flexibility in identifying potential concurrency (16, 23, 118). These issues point to alternative perspectives of concurrency based upon whether the identified tasks are asynchronous or synchronous, and whether they are deterministic or nondeterministic.

4.3.1 Mapping Applications to Architectures. The designer must at some point consider the target platform for implementation. The decomposition of the problem specification into its required data and control structure becomes much more difficult and critical in the parallel environment. In order to effectively take advantage of the parallel capabilities, the designer must decompose the original problem specification to allow for maximum parallelization. This decom-

position may have a multitude of alternative decompositions, adding to the combinatorics of the problem decomposition process.

The parallel environment is made difficult not only by the number of varying architectures and languages, but also by the need to capture the inherent concurrency explicitly (23, 101). The designer requires specific methods for mapping the application to the machine. This allocation and scheduling problem is recognized as an NP-complete problem (48). Because this mapping is an NP-complete problem, many of the approaches use constraints, heuristic functions, and bounding factors in an approximate solution (mapping) technique.

The mapping process is complicated by the possibility of the *cardinality variation* and the *topology variation* problems (26). A *cardinality variation* results when the number of processors for the parallel hardware architecture does not match the number of processes for the problem's software architecture. This becomes a problem when the application's identified processes exceeds the number of available processing elements, requiring the addition of a control structure to manage the scheduling of tasks. A *topology variation* results when the parallel system's communication structure does not adequately support the required communications and synchronizations. Overcoming this problem may also require the addition of a control structure for sharing communication channels. However, each of these problems can be overcome by restructuring the problem's specification and software architecture.

A given decomposition has embedded synchronization, control, and data dependency, which relate the various tasks. Most methods of parallel decomposition can be represented using a graph-theoretic approach to model the application. The graph-theoretic approach uses a task system graph, where each node represents a subtask of the overall problem. The relationships between the subtasks may represent the embedded synchronization, control, or data dependencies. The task system can be viewed as composing a software architecture. Each node brings about a state change of the system and the model may include the specific control and data flow of the problem.

Once a designer decomposes the problem into the finest grain tasks (level of abstraction), there are various composition strategies for maximizing the concurrency for a given hardware architecture. The composition methods group the subtasks in such a way as to reduce the complexity of mapping between the software architecture and the hardware architecture. This increases performance through an efficient parallel execution and balance of the workload. The bottom line of any decomposition/composition strategy is to maximize performance. However, the designer must also consider issues of the inherent scalability and order of complexity of the design.

4.3.2 Summary of Hardware Architecture Issues. Each parallel machine may implement a different optimal algorithm that reflects a different view of the application problem solution, exploiting alternative parallel aspects of the design. Software designers must therefore understand the constraints of the various hardware architecture models and how the problem design specifications relate to these parallel models. A transformation process must have available different architecture transformations that optimize the application specification for specific hardware platforms. Researchers have published several techniques and guidelines to support the hardware architecture mapping process and to fine-tune the algorithm and data structure design. For example, tasks may be identified, decomposed, partitioned, and allocated in order to reduce communication costs and to map to a given architecture more efficiently and effectively (26, 88). The details of this process are described in Section 4.5. The design process can be automated by formalizing the mapping and allocation techniques into specific transformations, which use processor-specific information in generating the transformations.

Similar techniques have been applied for years by compiler designers (42). The compilers define the formal mappings between a programming language and the machine code; however, the compiler may also include optimizations and machine-specific transformations. These additional transformations apply simplification and restructuring techniques to improve the program's implementation on a specific target machine.

The design platform for this research considers the inclusion of automated techniques for parallel architecture mappings. More detailed information on the parallel software design issues and design techniques are provided in the following sections.

4.4 Issues of Concurrency

The previous section described how parallel systems seek to achieve speed-up through the concurrent execution of the processes that make up an application design. To accomplish this, the program implementation dictates a distribution of either the functional tasks or problem data between the processors. These alternative problem decompositions are referred to as, respectively, control decomposition and data decomposition (79, 80, 101). In addition, one may apply a combination of both techniques.

Each of the problem partitions represents an active process or task of the application's software architecture. The active processes may or may not need to interact and cooperate. Interactions between processes are needed to synchronize or exchange data. These specified interactions between the processes establish a dependence between the processes, which may limit parallel implementation.

A mapping of an application to a parallel hardware architecture is limited by these data and control dependencies. The dependencies establish a precedence among the processes or tasks, which may require a distinct ordering in execution to maintain correctness. The dependence may limit the amount of concurrent execution because of the required overhead communications and task execution precedence. Any scheduling of processes to processors must maintain the required ordering. The precedence may be maintained through communications between the processes, requiring a system level control structure. In a shared memory architecture, the process communication uses shared variables; in a distributed memory system, these interactions occur through message passing.

As stated previously, each method of communication has its own limitations and associated issues. The shared memory machines provide specific control structures in the system's memory management and operating system. The operating system and communication subprocessors of distributed systems provide the software designer with only limited assistance. The software designer must specify explicit communication commands for synchronization, communication, and data coherency.

The concurrency identified is usually implemented in one of two forms: either *true concurrency* or *pseudo-concurrency*. True concurrency occurs when the number of processors is equal to or greater than the number of processes; in true concurrency, actions occur simultaneously and independently on the separate processors. Pseudo-concurrency occurs when the processes are interleaved through a control structure and appear at a higher level to execute concurrently. Pseudo-concurrency may have a nondeterministic ordering that does not violate any precedence constraints, or it may have a specific control structure to maintain dependent activities (118).

Application problems vary in the ease with which they may be parallelized. *Embarrassingly parallel problems* are those which have a very easy decomposition into independent sub-problems. For example, calculation of employee pay depends only on information for the individual employee, so each employee's pay may be calculated independently. On the other hand, some problems have no readily apparent decomposition. The dependency between the given tasks requires a sequential calculation. For example, the calculation of the median of a series of numbers cannot be broken up into sub-problems. When there is no satisfactory control or data decomposition, problems are difficult to parallelize and may be restricted to sequential execution. These examples represent the two ends of a spectrum of parallelization. As described in Appendix B, many researchers have proposed specific computational laws and properties for describing parallel design issues.

The parallelism specified for a given problem falls between the two ends of the parallelization spectrum based on the dependence or independence of its tasks. Tasks that are independent can

execute concurrently without the addition of a specific control structure; on the other hand, if the tasks have a dependence they can no longer be performed concurrently without a specific control structure to synchronize their operation. For example, if a yard was maintained by a group of gardeners, one gardener could weed the flower beds, while another could simultaneously cut the grass. No communication would be required. However, if each task required the use of a rake and there was only a single rake, a dependency is created. The gardeners would need to coordinate their use of the rake. Another example would be if one gardener's task was to weed the garden, while another's was to water the garden. Some communication would have to occur to prevent the one gardener from getting wet. These examples are variations of the classic problem of the dining philosophers (23, 118).

The designer must evaluate the dependencies of the problem solution that place constraints on the concurrency. The methods of concurrent design attempt to distribute the work and minimize the dependence. Unfortunately, these objectives are often at odds. The overhead for communication and synchronization increases as the task is further distributed, and the dependency increases as the tasks are further decomposed and allocated among processors. Both factors limit the effective work accomplished, decreasing efficiency. The computational laws described in Appendix B can be used to determine the effective performance gains.

As identified in Chapter 1, parallel software design has been traditionally accomplished in two ways: parallelizing compilers or using explicit language constructs for parallel algorithm specification. The parallelizing compilers can be used to automatically restructure sequential programs for parallel execution. The parallel compilers are limited by the algorithms for identifying inherent concurrency. The automated support of operating systems and compilers is discussed below (See Section 4.4.1). The second method of achieving parallelism uses parallel language constructs to explicitly identify concurrency. The designer uses the explicit constructs to identify the inherent parallelism of the application. This second method is limited by the designer's ability to identify

the concurrency (7, 23, 50). The details of parallel design using explicit language constructs is discussed in Section 4.4.2.

4.4.1 System Support for Concurrency. Early operating systems research looked at problems of critical region, mutual exclusion, atomic action, control flow dependence, and deadlock (79, 118). To overcome these problems, researchers have provided improvements to the operating systems, additional control structures within the algorithms, and hardware support. The difficulties encountered by operating systems were due to the need for tasks to operate concurrently. Our earliest understanding of concurrency came through this operating systems research.

The solution to the critical region problem and data coherency problem have included the addition of atomic action commands, spin locks, barriers, semaphores, and monitors (118). These control structures ensure the protection of the critical region or shared variables. In many cases, concurrent operation could not occur without the use of these sophisticated control structures. These added control structures may be embedded in the operating system or computer hardware. The user is unaware of their impact on maintaining the correctness of concurrency for the implementation. Unfortunately, operating systems research needs to mature before the designer can rely on its complete support in parallel systems.

Also used in the support of concurrent designs are parallel compilers. Compilers attempt to add concurrency by performing dependency checks and parallelizing those portions of the algorithm that are independent. In order to maintain the consistency and completeness, they may also require the addition of a control structure due to issues of control flow dependence, process synchronization, or data coherency. Because of the addition to the problem structure and the possible restructuring of the problem to create parallelism, these *sophisticated* compilers are also referred to as *restructuring compilers.* The compilers automate the process and abstract the need for the designer to explicitly identify the concurrency and add the control structures to maintain the correctness. But the automated process of the compiler is limited. The compilers take a conservative

approach, potentially missing possible inherent parallelism. Examples of compiler transformations include variable renaming, program unfolding, and loop unrolling.

4.4.2 Parallel Algorithm Design. Experience has shown that the use of compilers to restructure sequential algorithms has not been completely effective. An alternative to using the compiler is using explicit language constructs. The lack of extensive knowledge on parallel issues limits this method. The designers apply informal techniques of trial and error to gain more knowledge about the problem and its parallel implementation. As more knowledge is gained about a problem solution's inherent parallelism, the specific process of design becomes simpler.

Many researchers are now evaluating and developing new techniques for parallel algorithm design using explicit parallel language constructs. In many cases the computational problem must be reanalyzed, and a completely new parallel algorithm developed. There has been a considerable amount of research to solve a range of problems including sorting, search, graph processing, solutions to linear and differential equations, and simulation (79). One can refer to any of several available textbooks to learn about available algorithm design methods and obtain existing algorithm specification that explicitly identify the parallelism (18, 38, 45, 46, 61, 63, 68, 69, 79, 100, 101, 118).

The general parallel design methods that are applied by researchers can be categorized as follows: Data Parallelism, Data Partitioning, Relaxed Parallelism, Synchronous Iteration, Replicated Worker, or Pipelined Computation (79). Each of the parallelization methods takes advantage of a certain aspect of the parallelism of application under development or the available processing system. These particular methods are essentially specializations of data or control decomposition, or a combination of the two. As stated previously, the concurrency limitations are due to dependencies between tasks.

The early work on concurrent programming focused on distributing *the threads of control* of a process, providing multiple threads of control. The splitting of process control was explicitly specified using a *fork* command, and united through a *join* command. The newly created process

became known as a *child*, and the creating process a *parent*. The synchronization between the tasks is achieved through a join and may require either process to wait for the other (118). The level of parallelism during this early research was at the procedural or program level, providing primarily coarse-grain parallelism.

The use of parallel constructs has evolved with new languages using explicit commands, e.g., parbegin and parend, cobegin and coend, or DOALL and DOACROSS (91). Although the syntax has changed, the parallelism is still achieved through multiple threads of control (concurrent sequences), each with starting and terminating point for parallel activity. The granularity of the level of parallel activity may vary in the methods used today, to include parallelism down to the instruction level (fine-grain).

The algorithm designer uses the available constructs of the language to specify the composition of statements that can execute concurrently. The level of concurrency may vary depending upon the inherent control structure of a program. The specification of the concurrency may additionally require varying levels of control structures in order to maintain the desired consistency and correctness. For example, multiple assignment requires the actions to occur simultaneously, whereas other forms of concurrency may allow a nondeterministic ordering. Multiple assignment is a method of specifying multiple actions to occur simultaneously. Therefore, through multiple assignment these actions can be viewed as a single event and single state change. Unfortunately, the computer cannot accomplish these actions simultaneously and must provide a control structure to maintain correctness in operation (such as the operating system techniques defined previously).

Other properties of problem identification and specification may also contribute to the inherent concurrency. The benefits of nondeterminism in concurrent programming were discovered by Dijkstra. Dijkstra found that one could introduce nondeterminism into a program specification and control the correctness of execution through the use of *guarded commands*. Nondeterminism allows for greater flexibility in creating possible concurrency in algorithm design and avoids some of the

control structure problems associated with a multiple assignment approach (35, 44, 118). In addition, the nondeterminism allows for the active processes to be interleaved for a pseudo-concurrent execution.

4.4.3 Summary of Parallel Programming Issues. Software designers must create the specific control structures and data structures to define the application solution. The concurrency of the problem must be captured within these structures. The requirements for parallel algorithm design need to be incorporated into the software specification and modeling process. The application control structures define a specific approach for meeting the functional requirements of the problem. A correspondence between the functional requirements and the defined control structure must be maintained to ensure execution correctness. In other words, the operational semantics for the concurrent specification must include the required control structures to maintain consistency and completeness for the behavior of the intended solution in execution.

The designer creates the control structure that defines the algorithm and specifies the data structure interactions. The control structures play an important role in defining the inherent dependencies of the software architecture and its inherent concurrency or parallelism. The control structure must maintain specific assumptions (implicit knowledge) about the architecture model as well as any explicitly defined precedence constraints of the problem. At a system level, the control flow has an inherent sequence of activities and the control structures must maintain this sequencing. By specifying a problem to allow for nondeterminism, the system's activities can be interleaved in a pseudo-concurrent representation. Through a specific mapping to multiple processors, the designer can achieve concurrency.

In applying these concepts to the proposed development environment of Chapter 3, the software architecture models must specify the active processes of the problem and their interactions. The model must capture the specific requirements for the dependencies of the execution flow. In generating the model, the designer should seek to incorporate possible nondeterminacy and task in-

dependence into the problem specification. For this purpose, the parallel specification environment and its underlying parallel language, the software architecture model, and the domain model must all include concurrent properties. The operational semantics for the language must address the concurrent issues of the algorithm design and the architecture of the underlying parallel machine.

4.5 Task Management

Task management encompasses all aspects of parallel design and implementation. The focus of task management is to maximize the efficiency and utilization of the computer resources while minimizing the execution time and schedule for a given task system. Task management includes a decomposition of the problem into individual tasks, an allocation or partitioning of those tasks to processors of the system, a partial ordering and synchronization of those tasks into a schedule (scheduling), and a balancing of the workload across the system (load balancing).

The term *task scheduling* usually encompasses both the allocation and scheduling and thus is an assignment of tasks to processors and a synchronization between tasks. On the other hand, *load balancing* focuses on a means to create an equal distribution of work throughout the system. Both concepts focus on improving the overall performance of the computational model. Because of the mutual emphasis of each of these areas on performance, there is only a subtle distinction between these terms. For example, an effective partitioning strategy can achieve an adequate static load balancing for a given problem. For an additional analogy, the load balancing may alternatively be viewed as a dynamic rescheduling of the system tasks.

An efficient task management strategy is important for obtaining an efficient parallel implementation. The task management techniques provide informal methods for generating the process scheduling through a mapping of tasks to processors. The schedule must maintain the task precedence defined for the application. Advances in our understanding of the overall scheduling and mapping problems have provided improved heuristics for approximation algorithms. The techniques

also provide a means of overcoming the difficulties of the topological variations and cardinality variations described previously (see Section 4.3.1).

In the same manner that a business cannot be run effectively without proper management, the design of a parallel application's computational model requires the proper management strategies. An effective and efficient implementation of a parallel application cannot be achieved unless the inherent problem concurrency is appropriately identified in the problem decomposition process, the problem communication overhead is reduced, the efficient use of system resources is obtained through the allocation process, and the execution precedence between tasks is maintained for problem correctness in the scheduling process. The proposed conceptual software development environment must include techniques and tools to support a comprehensive task management process.

This section provides a description of the task management issues and relates them to the proposed development environment. In addition to this material, Appendix E provides a description of specific algorithms and automated tools to support the task management process. The appendix describes the current capabilities of several task management techniques that could possibly be incorporated into the proposed environment.

As mentioned previously, Appendix D provides additional background and analysis of the task management process. A reader unfamiliar with scheduling and load balancing problems may wish to read this material before continuing.

4.5.1 Problem Decomposition. Effective use of a parallel computer requires decomposition of a problem into subtasks that can be processed concurrently and then readily combined into an overall solution. The subtasks are associated through their data and control dependencies. Problem decomposition within the parallel software development process incorporates many levels of design. Initially, a decomposition is performed in domain analysis to identify the entities of the problem. The objects and operations identified through this process are then used to compose a model of the problem. The model specifies a structural relationship between the entities and operations of the

problem and may alternatively be referred to as a software architecture model of the problem. In a parallel environment, this model must then be further decomposed and mapped to the processors of the parallel system.

As mentioned previously, there are two basic problem decomposition strategies for exploiting an application's inherent parallelism:

- Data decomposition, in which a large data set is spread over many processors. Designers use data decomposition in problems where the same calculations are performed over and over on different data. Each processor is assigned the same functional task and processes a specific partition of the data.
- Control decomposition, in which the algorithm is broken into sub-algorithms to be spread over many processors. Designers use control decomposition in problem areas where multiple independent subtasks can be performed concurrently. Each processor is assigned a different task of the overall problem.

The two strategies are not mutually exclusive, and some problem decompositions use a mixture of both strategies in mapping to the processors to increase problem concurrency. Both methods generate a decomposition of the problem that results in a structuring of the application's active processes or tasks. The emphasis of the parallelization of the problem is on the decomposition in support of a specific assignment to the processors.

The decomposition strategies for the exploitation of the application's parallelism are at a different level than the design strategies used for decomposing the application. For example, the designer may apply an object-oriented, functional, or logical decomposition in the initial identification of the problem. These decompositions are in support of a specific design strategy that defines the encapsulation of the program code and data through a specific domain analysis.

In the software development environment proposed for this research, the domain analysis applies an object-oriented approach to define and decompose the objects of the domain. To define

a specific problem instance, the designer must additionally define a specific data and control flow model of the problem to act as a program driver and define the interactions of a computational model. The designer composes a software architecture model of the problem using the objects and operations identified in the domain analysis. The specific architecture model generated must define the specific control and data flow. To parallelize the problem, the designer must then apply a specific decomposition of this *driver* model to support an allocation to the individual processors. The designer accomplishes an effective overall parallel strategy by establishing a relationship between these different design decomposition levels.

Each task is an active process that results in a series of operations that change the state of the machine through modified data. The required control structure identifies the synchronization and data transfer requirements and will vary based on the encapsulation strategy used. A task system graph is a common abstraction of an application. The problem decomposition is represented by the individual nodes (or tasks) of the graph. Applying alternative parallel decomposition strategies will result in different task graphs with different inherent concurrent properties.

Using the task graph abstraction, the result of the decomposition is the identification of a set of tasks and dependencies. The tasks are modeled as the nodes of the graph and the dependencies as the arcs. Granularity is a measure of the size of the tasks in relation to dependencies. It is important to the overall mapping process to establish a correspondence between the granularity of the problem and the granularity of the target parallel system. In addition, the uniformity of the grain sizes of the individual tasks will have a direct impact on the implementation performance through the available mappings. Specifically, the load balance and scalability of the problem are directly related to the problem grain size.

As discussed previously, the dependencies between tasks limit the effective parallelism available through the architecture mappings. Concurrency is achieved through a mapping of a set of independent tasks to the parallel system. Therefore, it is important to maximize the independence

of the identified tasks of the problem, although this may additionally impact and unbalance the overall granularity of the problem. The designer must balance these trade-offs by evaluating the different decompositions and their mappings. The design may apply specific compositions to group tasks to balance the granularity and reduce inter-process communication (dependencies).

In summary, the overall problem decomposition would be a combination of the domain analysis process and the designer's interactive software architecture composition. The domain analysis would populate the knowledge base with the required objects, operations, rules, and constraints. The designer could then use the composition system to generate a task system graph for the application. Encapsulation of the objects would be interactive, allowing different compositions through the domain-specific transformations and rules of composition. The created software architecture would then need to be decomposed in support of a mapping to multiple processors for a specific parallel implementations. The architectural mapping tools of the environment must support a (re)structuring of the software architecture model to facilitate the mapping to a target parallel system and to optimize performance.

4.5.2 Partitioning. Partitioning emphasizes the locality of the tasks and the data for their allocation and mapping to the hardware architecture. The designer creates specific groupings of tasks to minimize interprocessor communications across the system or to build some inherent structure within the design. The partitions attempt to balance the workload across the system. For static or regular problems, information about the control and data flow is known *a priori*. Thus, regular problems can be distributed to the processors with an even load balance and reduced communication through the process assignment and scheduling. For dynamic or irregular problems, the trace information for the control and data flow is not known until execution time. The designer must use an approximation technique to distribute the processes. The distribution is usually combined with a dynamic load balancing strategy built into the system control structure. The load balancing scheme allows specific tasks to migrate between processors to achieve better efficiency.

Compilers also perform partitioning to allocate the required data to memory locations for the effective and efficient utilization of the processors. Other techniques use partitioning within load balancing strategies, where the underlying processor architectures have been partitioned through some categorization. The categorizations support a functional decomposition of the problem. The designer assigns specific types of tasks to specific processors.

The partitioning of application problems for parallel design combines aspects of problem decomposition and task allocation. Therefore, partitioning concepts may be contained in both the decomposition and scheduling of tasks. Some applications do not identify a separate partitioning step of design because of the strong association to decomposition and scheduling. In addition, because some partitioning is performed by the compiler, and is therefore transparent to the user, it does not receive the attention of the designer.

An example of the use of partitioning can be seen in the applications of a fluid dynamics problem on a matrix computer. The designer may use partitioning to group the processes together, thus grouping tasks, and allocate these groups to a specific section of the matrix grid. The processors on the borders between sectors must periodically exchange data; however, the internal processors of the sector can continue processing.

4.5.3 Scheduling. The scheduling, assignment, and mapping problems have been extensively researched and studied in operations research. Task scheduling in the parallel environment combines operations research techniques and graph-theoretic techniques to define and model the application. The specific task precedence is captured and used in heuristic algorithms to create and compare alternative schedules. Because of the NP-completeness of the scheduling problem, task scheduling usually applies approximation algorithms.

The overall problem focuses on a distribution and allocation of the tasks throughout the parallel system to maintain task precedence. The system schedule's control structure must maintain

a partial ordering of the execution of tasks in the parallel system. The partial ordering maintains the task dependencies and precedence so as to preserve the correctness of the desired behavior(s).

As identified in Appendix D, task scheduling within a parallel system occurs both locally, on each processing node, and globally, across the system. A specific control structure may be required to maintain the precedence between tasks at each level. For example, specific synchronization variables may be added to ensure that a specific task A on processor 1 executes before a specific task B on processor 2. The requirements for this precedence must be represented in the initial problem model in order to be incorporated into the schedule.

Task scheduling techniques can be classified as static, occurring at compile time, or dynamic, occurring at execution time. The static methods are applied to regular problems, where knowledge about the application's execution is known *a priori*. The dynamic methods are applied for irregular problems, where little is known about the application's execution. The dynamic methods are closely associated with the load balancing concepts described later. The optimal schedule focuses on minimizing the execution time for a given application. An effective schedule must maintain the partial ordering and synchronization of the tasks for a given execution.

Most of the available research on scheduling tools focuses on static scheduling. Unfortunately, this requires the *a priori* knowledge of execution time, precedence, and execution trace; these parameters are not readily available for most real world problems. However, through approximation techniques, better problem understanding, and sophisticated load balancing, the application of the static techniques on both regular and irregular problems may be possible.

Dynamic or irregular problems, with a control flow dependent upon the input data, offer a challenge for task scheduling. Unfortunately, due to the combinatorics and dynamic nature of the execution of these problems, there is no realistic way to optimize these problems for all instances. Designers must rely on domain-specific knowledge and parameters for approximation techniques. Task scheduling for the dynamic problems must be performed in conjunction with the operating

system control structure and load balancing strategies. The designer performs an initial static allocation and partitioning of the tasks to the architecture. As the execution progresses, adaptive methods are used to migrate the tasks and reschedule processes within the system. The dynamic load balancing must still maintain the task synchronization precedence and the data and control dependencies.

Additional information on task scheduling is provided in Appendices D and E. These appendices provide greater detail on the specific terminology and provide examples of specific algorithms and tools. The parallel software development environment proposed in Chapter 3 will require appropriate tools to transform the formal specification of an application into a specific implementation. The tools described in Appendix E provide a starting point for defining the requirements for these task management tools.

4.5.4 Load Balancing. Load balancing techniques range from static to dynamic. The techniques focus on maximizing the efficient utilization of all processors throughout the system. Static load balancing principles are usually incorporated into the heuristics of the task partitioning, allocation, and scheduling algorithms. The control structure of static load balancing is predefined at compile time. Dynamic load balancing uses system information to redistribute (and thus reschedule) tasks within the system at runtime; therefore, dynamic load balancing is usually used in conjunction with dynamic adaptive scheduling.

The designer chooses the strategy based on an understanding of the constraint of the specific application. A detailed description of various methods of load balancing is discussed in Appendix D. Load balancing strategies require the creation and integration of a specific control structure for managing the tasks on the local and global levels. The designer must evaluate the trade-off between the additional overhead processing time to generate a more uniform balance and the benefits gained by this greater efficiency.

The objective of load balancing is to maximize the overall system performance for a given application. Because of the strong affinity to the system performance parameters, designers usually incorporate features of load balancing within problem decomposition, partitioning, and scheduling strategies. Another method of balancing the processor load is the use of multi-tasking, taking advantage of the operating system and hardware support. As discussed in Section 4.5.3, load balancing applies a reallocation or rescheduling of tasks and must maintain the underlying synchronization and precedence constraints. The constraints can be maintained via a more sophisticated control structure and partitioning strategy.

The incorporation of load balancing strategies into the parallel software design environment proposed in Chapter 3 would be accomplished by the creation of specific theories in the knowledge base. These strategies could then be applied during the transformation process on an application so that they could be incorporated into the problem's formal specification.

4.5.5 Task Management Summary. In order for the automated design process for the parallel software design environment proposed in Chapter 3 to be effective and efficient, a comprehensive set of architecture mapping tools is required. The composition of tasks forms a software architectural model of the logical processes of the application that could be modeled through the graphical techniques described in Appendix E. The hardware architecture can be represented by a graphical means as well. Alternative mappings and task groupings can be evaluated through an interactive environment.

The encapsulation of the formal representation of the tasks of the system is abstracted through the graphical representation or *task graph.* Existing algorithms and tools can be applied for this environment to manipulate the task graph and map it to the architecture for efficient execution. The parallel composition constructs must support the creation of the task graph abstraction and its manipulation. This places specific commutative and associative requirements on the definition of eventually parallel operators.

4.6 Chapter Summary

The design of software within the parallel environment faces many challenges. At times the goals of design may be at odds, requiring an evaluation of the trade-offs when managing the tasks. The designer seeks to maximize parallelization, while dependencies limit the amount that an application can be parallelized. Load balancing tries to distribute the workload evenly among the processors, while reduction of interprocessor communication tends to lead to assigning the workload to a single processor. Maximizing the number of tasks that can be executed concurrently may overwhelm the execution by overloading the required amount of communication. Each of these issues must be addressed in the development process.

Current parallel design techniques focus on optimizing the software to efficiently run on a given parallel hardware architecture. The optimizations try to reduce the complexity of a mapping between the software architecture (logical processes) and the hardware architecture (physical processors) and maximize the performance efficiency through reduced interprocessor communication and execution time. Parallel implementation focuses primarily on performance; therefore, the inclusion of implementation and optimization techniques are important for a parallel software development environment.

This chapter presented the issues involved with creating the mapping between the software architecture and hardware architecture to generate an implementation. It described the characteristics of parallel architectures and the problems they face in the concurrent environment. The alternative design issues of the multiprocessor and multicomputer were also identified and described. The transformation process must address these alternatives in the mapping to an implementation.

Parallel software and algorithm design face many concurrency issues, requiring a comprehensive parallel design strategy; the traditional sequential design methods may not apply. Hardware and software design methodologies are gradually changing to incorporate parallel concepts. The

use of sophisticated control structures for addressing the problems of concurrent execution have become standard.

The task management techniques presented in this chapter provide an example of the comprehensive strategies being applied to parallel design. The task management techniques incorporate all aspects of parallel design. Using a task management approach, the design process focuses on a management of the active processes (i.e. tasks) and their efficient interaction. This technique abstracts the specifics of the individual tasks, allowing for an effective and efficient mapping to a parallel system.

Explicit language constructs for parallel representation are incorporated into many of the programming languages. The language constructs created to define a design's explicit parallelism require the incorporation of a specific control structure within the environment to maintain their operational semantic. The definition of a complete set of parallel constructs requires the consideration of the issues described in this chapter. In addition, the issues of concurrency highlight the differences between asynchronous and synchronous parallel execution and their relationship to deterministic and nondeterministic execution. A complete set of parallel constructs must have the capability to represent each of these properties.

The optimization techniques used in the informal processes of current parallel design can be incorporated into formal optimization transformations in the proposed research. The development of the envisioned platform described in Chapter 3 requires the inclusion of tools to support transformations of the specification to optimize execution. The information included in this chapter and in Appendices B, C, D, and E detail the issues involved in parallel software development. The goal of this chapter and these appendices is to both provide the reader with the appropriate background on the issues of parallel design and identify the issues that must be addressed in defining a complete set of parallel language constructs.

Thus far, the impact of the hardware, software, and problem specification have been described to characterize *optimization* and *implementation* issues. The next chapter describes several existing formal specification environments that the designer can use to formalize the problem description. The formalization of the application generates a description of the problem solution that can be used in the mapping process described in this chapter.

V. *Description and Analysis of Formal Specification Languages and Environments*

5.1 Introduction

This chapter describes several existing specification languages and environments. Key concepts and features for syntax structure of each language are discussed. The evaluation of these languages focused on the availability of explicit concurrent constructs. The assumptions and implicit concurrency issues were also analyzed. In addition, the formal semantics were used to determine each language's capabilities and its applicability to the conceptual design of parallel software design environment proposed in Chapter 3. The chapter concludes with a summary of the language features.

5.2 Formal Specification and Design Environments: Description

Most of the languages evaluated incorporate aspects of first-order predicate calculus and set theory. The languages share key concepts (semantics) while using different naming and structuring (syntax) (52, 54, 67, 56, 108). In addition, the available support tools have common objectives and functionality.

The following sections describe the different specification languages and environments. The languages included for evaluation were selected based on a literature search of specification languages for parallel programming, transformational programming, and formal software development. Each section describes a language's objectives, its formal syntax and semantics, its application of concurrent constructs, and its characterization of concurrent features.

5.2.1 Parallel Virtual Machine (PVM). PVM is a software package that allows a heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource. Research on PVM began in 1989 at Oak Ridge National Laboratory (ORNL). The current version, 3.1, includes additional extensions provided by ORNL with the support of a University

Consortium. The purpose of PVM is to abstract the machine (computer) and its associative implementation control structure from the program designer (49).

PVM consists of two parts: a daemon process that any user can install on a machine, and a user library that contains routines for initializing processes on other machines, for communicating between processes, and for changing the configuration of the machine. The daemon process acts as an extension of the operating system to interface between the machine control structure and the application program. The PVM user library provides additional language constructs to support process synchronization, communication, control, and network configuration/reconfiguration.

Unlike other design environments evaluated, PVM is not a formal specification environment; however, PVM provides a virtual environment for concurrent programming that allows for machine-independent design. PVM requires machine-dependent and operating system specific daemon processes installed in the computer in order to operate correctly. The power of PVM, and its applicability to this research, stems from the uniform view provided to the programmer through abstraction of control structures.

PVM *does not* provide enough of the required formal properties or transformational capabilities required for the parallel software design environment envisioned by this research. Therefore, PVM is not considered for inclusion within the proposed environment. However, PVM does demonstrate a design approach that abstracts the specific control structures of the machine from the specification and highlights the importance of this feature for design portability.

5.2.2 Formal Specification Language GIST. GIST is a specification language developed at the Information Sciences Institute of the University of Southern California (9, 82). GIST is a textual language based on a relational model consisting of objects, attributes, and relations. The mathematical basis for the language provides for the formality of the constructs. GIST is a wide-spectrum language, containing both high-level specification constructs and constructs of existing low-level programming languages (94).

GIST was developed to "provide the flexibility and ease of expression necessary for describing a full range of acceptable behaviors." Its wide-spectrum capability was added to provide for not only a reasonable specification of a problem, but also for an implementation that is consistent with the specification (93).

The GIST environment's support of automated programming stems from the inclusion of the following concepts (10):

1. Database

   The GIST environment contains a powerful database for the storage of the specified objects, relations, and operations. The GIST language includes constructs for the manipulation of the database with uniform accessibility at the specification level. The database is also referred to as the knowledge base.

2. Operational Environment

   The GIST language uses operational (executable) semantics. The various constructs can be used to describe and display the software architecture's behavior. The user provides the description in terms of concurrent interactive agents. The description attempts to formally define the system, ignoring the need for efficient execution.

3. Perfect Knowledge

   Inference and derivation rules can be included as needed (based on established conditional triggers). In addition, implicit knowledge can be embedded into the knowledge base.

4. Descriptive Reference

   Relative addressing of the objects is provided, allowing access to objects through their attributes and relations.

5. Historical Reference

   The trace history for the transformation process can be stored and re-accessed or rolled back. The user can also reference previous state values.

6. Constraint Avoidance

   The use of a subset of behaviors as a set of constraints allows the system to operate without violating the invariance (Note: constraint specification is similar to the *always* feature of UNITY; see Section 5.2.6).

7. Closed System

   In order to define the interaction of the specified system within its environment, that environment must also be specified. By additionally specifying the environment of operation in GIST, the entire specification can be simulated and analyzed.

These principles were included in the design of the GLITTER and PADDLE programming environments (10, 13, 41). These environments were based upon the transformation implementation (TI) model of software development (41) using the GIST language. The environments apply automated techniques to transform abstract specifications using GIST into more concrete representations of the problem. Specific programming and domain knowledge interact within the automated environment to synthesize the concrete specification. The environments provide an execution capability for the GIST specification, providing immediate feedback of the specified behavior(s).

The GIST specification captures the valid behaviors of a system by including the following (94):

- A specification of the object types and relationships between them,
- A specification of actions (transitions of a state model), demons (which trigger actions), and agents (composed of demons and actions)
- A state specification of constraints and transitions.

The system designer captures various aspects of system behavior in specification schemas that include keywords for defining an object's type, attributes and relations, as well as any actions of the object. An *action* changes the state of the respective process or task. Included within this model

are primitive functions: *create/destroy* for creating or deleting an object instance, *insert/delete* for modifications to an object through inserting or deleting constraints or relations, and *update* for updating the changes. A *demon* provides a mechanism for data-directed invocation of processes (94). A demon has a trigger and a response. The trigger is a predicate condition, which invokes a state change response whenever transitioning between true and false. For example, a response might be the creation of a new object or action.

Constraints are provided within GIST to specify and maintain integrity conditions. Constraints are specified through the use of the keyword *always*, the keyword *prohibited* or *required*, and a predicate condition. Constraints establish the invariant parameters of the system and provide deterministic attributes of the system. Constraints may be global or tied to specific actions or triggers.

For the specification of concurrency, GIST has the concept of *agents*. An agent is composed of a collection of actions and/or demons. The set of actions provides for the specific agent's capabilities, and the set of demons dictates how the agent responds to state changes of the process. Grouping the processes together within the agent(s) allows for their concurrent activity. This feature allows for the composition of concurrent tasks, but the structure of this encapsulation does not facilitate the mapping to fine-grain parallel architectures (94).

GIST played an important role in spearheading research into knowledge-based software engineering and automated environments. Its contributions to the research field can be seen in the design of other specification languages and environments created since its conception in the early 80's (for example, the development of REFINE and CIP-L included concepts identified and lessons learned through GIST's development and use). However, GIST does not have the commercial or academic support it once had. Languages such as REFINE and CIP-L provide improved environments for formal software design and a greater diversity of defined constructs.

5.2.3 Software Refinery Specification Environment. Reasoning Systems, Inc. was founded in 1984 in order to apply research concepts of knowledge-based programming toward the creation of a commercial product. In realization of this goal, Reasoning Systems developed a commercial knowledge-based software environment called Software Refinery, which includes a programming language, REFINE.

The REFINE programming language provides a range of specification features, characterizing it as a wide-spectrum specification language (103). The range of available constructs includes very high-level constructs, such as setformers and transformational rules, as well as conventional procedural constructs, such as *if-then-else*. REFINE includes concepts of set theory, logic (rules), procedures/functions (maps), and transformations (103). The REFINE syntax is built upon an underlying schema language. The REFINE constructs are defined using the schema language and encapsulated in a LISP package.

The data types supported by REFINE include: integers, reals, characters, symbols, boolean, sets, sequences, strings (sequence of characters), tuples, maps, binary relations, and objects (user defined). Each of these data types has predefined operations available. These structures are merely the *pre-defined* object types that are delivered with the environment; the user can define more complex types using these primitive types. The formal syntax of these structures is defined using a lower-level schema language. Because of the flexibility provided for defining the objects, REFINE does not limit the user's choice of data structures. Thus, Software Refinery also provides the ability to extend the available data and algebraic types through the use of the Dialect support tool and language constructs (103).

A REFINE user can define and manipulate objects, which include object classes, types, functions, and grammars (103:1-4). This feature allows the user to define domain-specific languages due to a mapping between the domain-specific variable names and the domain-specific objects. REFINE also provides support for a wide range of programming paradigms, with the exception of

concurrency. The designer can use the tools of the environment to add constructs and transformations incorporating domain-specific knowledge, adding to the clarity of the problem description (103).

Besides extending the REFINE grammar, a user of REFINE can define new objects and types based on primitive objects. The new objects inherit the characteristics of the primitive objects used for their definition. The relationship structure of the newly defined types can be viewed as a hierarchy of objects stored in an object base. Through this hierarchical defining of the language constructs, the designer can exploit encapsulation, inheritance, and abstraction in the design. These features allow the application of object-oriented design principles. However, the specific inheritance does not transcend multiple levels.

The REFINE language provides procedural control constructs such as while loops and enumeration over sets and sequences, which further reduces specification complexity. These control structures, combined with the transformation rules and mappings for state-based behavior, provide a breadth of design description. In addition, delimiters provide a means of grouping statements and expressions to control the execution flow through the program. For example, statements may be grouped by using a ";" between consecutive statements. The ";"signifies a connection between the statements, referred to as a *block*. This grouping provides a sequential composition of the statements and expressions. The operational semantic for the block's execution is provided by selecting a specific sequence for the statements' or expressions' execution.

The delivered REFINE environment does not include concurrency constructs; however, users of REFINE might exploit and expand the existing features to define concurrency. Specifically, one could extend the language to add concurrent composition, nondeterministic choice, and execution fairness. The user can add these features using the tools of the REFINE environment to define new constructs of the language.

REFINE's execution capability provides a unique environment for formal specification of application problems. Correctness proofs can be performed not only through pencil and paper calculations, but also through an executable behavior validation. A REFINE user can compile and execute a program specification, observing the exhibited behavior. The task of refining the specification to a low-level procedural implementation is performed automatically by the compiler based on a specific operational semantic. The compiler does not optimize the REFINE specification for efficient execution. REFINE ignores the efficiency of execution, focusing on behavior correctness.

There are additional tools that were designed to support REFINE. These tools add to REFINE's flexibility and effectiveness as a specification language. The addition of an object base representation (an abstract syntax tree model), a parsing tool, and object-oriented constructs contributes to the power of Software Refinery.

The REFINE environment lacks performance feedback and parallel architecture mapping tools for the implementation on a variety of parallel architectures. Specific support tools for the optimizing the program description and mapping to a parallel architecture are not provided, though the *rule* and *transformation* constructs already include powerful features toward defining optimization transformations required for these tools. A library of transformations could be generated for the optimization of REFINE specifications. A specific tool could be designed using defined transformations for creation of a parallel mapping tool.

In summary, the REFINE environment includes many attractive features for a knowledge-based, transformational programming environment; however, the delivered system lacks concurrency. The environment does provide a means to extend the language to add new language constructs and structures. This would be useful in applying REFINE to the parallel software development platform proposed in Chapter 3.

5.2.4 Computer-aided, Intuition-guided Programming (CIP). The CIP project began in 1974 at the University of Munich (54). The project evolved from a desire to create a unified semantic

basis for a transformational programming methodology using a stepwise refinement process. The goal was to formalize the design process, thus reducing the complexity of software development. The primary activities of the project, outlined at the Second International Conference on Software Engineering (15), are:

- Design and formal definition of a wide-spectrum language, CIP-L
- Design and implementation of a system, CIP-S, for program transformation
- Development of a methodology of inferential programming

The first attempt aimed at a layered system of languages of different levels of abstraction and alternative operational semantics. After further analysis, the designers abandoned the layered approach for a single language, covering a wide spectrum of styles. The designers required a language that incorporated both specification semantics (denotational and axiomatic) and operational semantics to support the entire transformation process (see Section 2.3). In addition, the language needed to support variations in programming style (e.g., functional, logic, and object-oriented). The designers identified the following advantages to a wide-spectrum approach:

- The wide-spectrum language allows for local changes of style, allowing incremental builds, without affecting the entire software design.
- Correctness proofs are localized.
- The semantic basis for the entire language gives complete freedom in the transformation process — transformations can be localized.

The variations in programming style also required a change in methodology and semantic frame. The thought process of the designer performing a decomposition of the problem in support of logic programming is much different from that used in an object-oriented approach. The designers concluded that through proper language definition and appropriate formal transformation rules, alternative variations in style could be supported. They generated a small kernel of constructs

containing the full expressive power of the language. One could then define other levels relative to this kernel through transformation rules, reducing all constructs to the kernel constructs (96). Thus, a hierarchy of knowledge was generated through the constructs and transformations from which inferential programming could be performed on this *knowledge representation.* To support such an approach, the language required the support of a sophisticated design methodology. The following additional requirements were identified:

- The language has to comprise a large number of different concepts, styles, and constructs. These constructs should allow the formulation of problem specifications and high-level programs as well as machine-oriented programs.
- During development, parts of the program may be incrementally replaced by functionally equivalent lower-level constructs, while other sections of the program remain unchanged. Thus at any one time, the program can contain a mixture of constructs without any syntactic or semantic constraints.
- Although the language supports different styles, the styles should be closely related. Transformation rules must exist between the various styles.
- The language should contain a formal framework in which formal correctness proofs may be applied to the transformation performed. Thus the correctness of the derived programs and their relationship to the original specification may be mathematically justified.

CIP-L provides specification and implementation constructs for both data and control structures. Included within the language are constructs for predicate logic, typed set operations, comprehensive choice, and general description. A program using these constructs is called *pre-algorithmic.* In order to generate an algorithm, the designer organizes these constructs into a control structure, setting up precedence and synchronization. The designer applies these primitive constructs using a chosen design strategy to generate a program in algorithmic form (i.e., to create an algorithm).

Thus strategies using recursion, conditional branching, and logical or functional abstraction are used to generate the concrete algorithms for the application.

The designers of CIP-L strove to separate the data (*algebraic types*) and control (*scheme language*) structures to facilitate a structured refinement process. A program under development evolves from abstract to concrete concepts. Data types are concretely defined through the algebraic constructs, with a mapping to concrete entities of computation. The *scheme language* focuses on symbolic manipulation of the program terms which are independent of the terms' meaning. The abstraction allows for freedom of design. The *scheme language* becomes more detailed as the design approaches the specific implementation version of the program. The transformation of scheme structure determines the implementation efficiency of the algorithm generated.

CIP-L is a fully typed language, with compiler support for static type checking. CIP-L provides a set of primitive types (boolean, integers, and floats) that can be expanded through a defined hierarchy. New types can be defined based on a relation to the existing primitive types. Although no specific list of primitive types is provided, the denotational syntax of the language allows one to build any of the types defined for REFINE (e.g., sets, sequences, maps) through its scheme language. The algebraic types are characterized through the use of algebra signatures and associated laws (axioms). One may also view these signature representations as similar to object classes, abstract data types, or Z schema representations. One specifies the associated sorts, operations, axioms, and constraints; they are captured in an encapsulated type specification to define the new types.

The ability to abstractly specify types and concepts, and build upon the kernel constructs, provides flexibility and expandibility for CIP-L. The formality of the low-level constructs used in building the hierarchical definitions of the new language constructs ensures the formality of the new constructs through the relational mappings. The distinction between what is and what is not CIP-L becomes a question of relativeness in specification. By loading the system knowledge base

with pre-defined types, one may expand the language to provide new abstract data types. The new types inherit the attribute of the types on which they are based.

One can view REFINE and CIP-L as almost equivalent because of the underlying mathematical basis (algebraic types, predicate logic, set theory, state-based theory) of both CIP-L and REFINE. The schema language capabilities of CIP-L are matched by REFINE's own extension and schema capabilities. Although CIP-L currently demonstrates more extensive features for support of concurrency, these features could be added and defined within the REFINE language using the extension capability of the DIALECT tool.

Because of the limited documentation and commercial support, a complete evaluation of CIP was not possible. Hence, CIP-L is only used for analysis purposes and not considered for future implementation.

The following features have been defined using CIP-L for support of parallel programming through shared variables: parallel blocks (*similar to cobegin* and *coend* of *concurrent* FORTRAN), guarded statements, await-statements (for blocking), parallel composition , and parallel procedures. These features were added by using the schema language to define new extensions. The definition of these parallel features required the addition of a new class of parallel statements.

The parallel statements include the addition of constraints or control structures. For example, conditions of *wait* (to prevent execution until a variable is defined and to synchronize), *protected* (to protect shared variables), and *non-conflict* (also to protect shared variables) are used to ensure that the proper conditions exist for parallel execution. These conditions are continually checked to ensure that the definition and execution of parallel statements are consistent with these properties. The ability to define these types of conditions (54:196) indicates the feasibility of adding the parallel control structures to REFINE's operational semantic.

The concurrent properties defined and maintained by these schemas provide a starting point for defining an operational semantic control structure for parallel composed objects. The defined

schemas, however, are incomplete. The explicit conditions do not account for synchronizations and communications required for the distributed environment. The current trend in parallel architecture design is toward a distributed framework (32, 47), and a complete description of parallel programming would require inclusion of distributed computer environments (i.e., multicomputers).

A complete parallel programming environment would require additional research to extend, refine, and update the defined schema. Our understanding of the parallel programming environment has evolved significantly since the publication of (54) in 1985. CIP provides extensive insight into on applying mathematical principles of algebraic types (and their use of mathematical signatures) and functional morphisms (transformations) to a software development process. In addition. the application of multiple semantic evaluation demonstrates the application of correctness proofs on multiple levels. Finally, the use of parallel schemas demonstrates some of the required constraints for software implementation and execution in the parallel environment.

5.2.5 Communicating Sequential Processes (CSP). In 1985 C.A.R. Hoare introduced Communicating Sequential Processes (CSP). The CSP language introduced a new method of reasoning about processes using language and grammar theories. CSP incorporated concepts of Hoare's earlier work on viewing processes as predicates (58). Each process could be abstracted by a predicate variable representing a state. The production rules of the CSP grammar become the state or event transitions for an abstract state machine description. The grammar notation is easily mapped to a graphical state machine description (57).

Hoare created CSP as a language analysis tool. CSP has been practically applied toward the development of several programming languages. For example, CSP is the basis for the Occam language. In addition, several researchers have used it for language analysis and construction of experimental programming environments (7, 56).

CSP uses various aspects of language theory for description and analysis. Because of the relationships between finite state machines (graphical representations) and grammars (syntactical

representations), CSP can capitalize on existing research. An application can easily be specified using either a grammar or state machine description. It can then be translated into the alternate method of description. Both approaches can be used to exploit and identify specific properties of the problem. For example, whether a ***fixed point*** is reached or whether a ***deadlock*** condition exists in the problem description can be detected using a graphical depiction and control flow analysis.

Fundamental to a system descriptions using CSP are concepts of recursion, choice, and trace. Recursion provides a means of defining a possibly infinite process through a finite description. Choice allows the branching between alternatives, introducing a branch in control flow and nondeterminism. Trace provides a finite description of a process as a series of events. The chain of events is associated with a sequence of time, where no two events occur simultaneously. Concurrency is introduced through the interleaving of traces. In addition, invariant constraints are specified through the use of *laws* (57).

The concept of trace plays a major role in the description of a process. The trace provides a specific control flow in the description of the process. Using language description operators and recursive functional descriptions for the process, one can finitely describe the process and a valid execution, as well as alternative executions. Specific operators are available for the logical manipulation of the trace, allowing for the logical analysis of different control flows or concurrent representations.

To understand the properties of concurrency within CSP, one must first understand CSP's trace rules concerning timing. The concept of time is abstracted from CSP through the trace operations. Events occur instantaneously as atomic actions and without duration. To describe the ordering of the events, the concept of a ***fair observer*** is introduced. This is an individual who observes the occurrence of events within the system and records them.

CSP provides two alternative representations for concurrency. The two methods are described as follows:

- When important events occur concurrently, they are recorded as a single event.
- When the event's order is not important, the events are recorded in an arbitrary order. The fair observer interleaves the description of these traces, providing a pseudo-concurrent representation.

Using the CSP timing rules, the concurrent processes are modeled through the use of either shared events or interleaved events. The shared events must be performed in lock step, whereas the independent events occur as interleaved traces with no dependency. The events themselves are control structures used to synchronize and communicate between processes. Thus, a shared event may act as a specific barrier for controlling concurrent synchronization and communication.

Although CSP provides a framework for describing processes through an abstract description, it does share the shortcomings of many of the formal languages in terms of understandability. The low-level description, combined with the multi-levels of abstraction makes it difficult to understand the direct correspondence between the syntactical model and the problem. The designer can partially overcome this through the use of mnemonic variable names with a direct relationship to specific terms, properties, and concepts of the domain. However, the complex nature of some problems make them difficult to describe in any language. The designer can only use specific structuring and descriptive techniques learned through experience.

5.2.6 Formal Specification Language UNITY. J. Mani Chandy and Jaydev Misra introduced the formal specification language UNITY in (23). They additionally described the refinement process for using this specification language and transforming the specifications for specific architectural implementations. The UNITY method of development provides a solid framework for design of parallel execution models, independent of the specific architecture. The backbone of the language is first-order predicate calculus and temporal logic constructs for correctness proofs. The language additionally provides explicit constructs for the representation of forced parallelism (e.g., multiple

assignment) and nondeterministic concurrency or choice. The ability to represent execution using a state transition model is a powerful feature of the language.

The UNITY syntax provides for simple representation of algorithms through a first-order predicate calculus representation. In addition, specific notation allows for the explicit representation of inherent parallelism within the algorithm. The language representation is architecture independent, allowing for transformational mappings to different architectures. The addition of detailed control constructs for detailing the specifics of *how* is left for implementation transformations. Chandy and Misra provide additional informal guidance for performing these mappings in the form of schemas. Thus a designer maintains a complete range of problem decomposition options (which is helpful when designing in the parallel environment).

The power of the UNITY approach applies stepwise refinement by abstracting the control flow until required for transformations to an implementation. A UNITY user applies extensive formal temporal logic proofs to ensure correctness. The concise nature of the proofs as they apply to invariance, fixed point. and progress provides an environment that enables nondeterminism, absence of control flow, and variations of synchronization from a platform independent perspective.

Nondeterminism is implicitly provided through the inclusion in the execution model of a *fairness* rule. The fairness rule is an extension of the temporal logic that establishes that an executable assignment statement will execute infinitely often and in an evenly distributed manner. The designer may choose the technique of guarded command to add nondeterministic choice (35).

The UNITY specification avoids the explicit specification of control flow for the initial high-level abstraction of the problem. Aspects of control flow are only added during transformation to implementation. The absence of this control flow in the early abstraction of the problem allows for greater flexibility in the later mappings to a specific implementation architecture and contributes to the specification of nondeterministic properties.

The UNITY specification allows for loose synchronization control. That is, due to the power of the temporal logic proofs, UNITY allows for a range of explicit and implicit synchronization specification. Based on whatever level is specified, determination of correctness is still feasible. This range of flexibility in synchronization control is essential in the parallel programming environment (23).

The language provides parallel (||) and concurrent ([]) constructs to explicitly capture the inherent parallel features. Applying the || composition combines a grouping of statements into a single atomic action. Applying the [] composition interleaves a grouping of statements.

The language additionally has high-level data structure features taken from the Pascal programming language and discrete mathematics for representation of arrays, sequences, and sets. Other structure can be generated by combining these with first-order predicate calculus. As stated previously, the UNITY programming constructs consist primarily of first-order predicate calculus, with temporal logic constructs used primarily for proof of correctness. However, the temporal logic constructs are not explicitly part of the UNITY language. They are used in defining an alternative axiomatic semantic of the specification for a proof of correctness.

UNITY can be viewed as using denotational semantics because of the separation of the specification from the implementation environment. UNITY does not provide an explicit operational semantic to define an execution behavior. Correctness of specification behavior is determined through a temporal logic proof process. The temporal logic constructs used for program proofs are defined as follows (this section has been extracted from the AFIT Compendium) (38):

5.2.6.1 UNITY Temporal Logic Constructs.

In the following descriptions, p and q are any two predicates describing the state of the program, F. The program, F, consists of a set of statements s.

unless. $\quad p \text{ unless } q \equiv \langle \forall s : s \in F :: \{p \wedge \neg q\}\ s\ \{p \vee q\} \rangle$

- Either p never becomes true at all, or
- p at some point does become true, and

- q never becomes true and p is always true, or
- q eventually becomes true, and p continues to be true *at least* until q becomes true.

stable. $\quad$ *stable* $p \equiv p$ *unless false*

- Either p never becomes true at all, or
- p at some point does become true and thereafter always remains true.

invariant. $\quad$ *invariant* $p \equiv$ (*initial condition* $\Rightarrow p$) $\wedge$ (*stable* p)

- p is true at the beginning of program execution, and
- p always remains true.

ensures. $\quad p$ *ensures* $q \equiv (p$ *unless* $q \wedge \langle\, \exists s : s \in F :: \{p \wedge \neg q\}\ s\ \{q\}\rangle$

- If p ever becomes true, then eventually q becomes true, and
- either p never becomes true at all, or
- p at some point does become true, and
  - q never becomes true and p is always true, or
  - q eventually becomes true, and p continues to be true *at least* until q becomes true.

Leads-to does not have a state transition definition as do *unless* and *ensures*, but its behavior can be implied from the above definitions. Specifically:

- $(p$ *ensures* $q) \rightarrow (p \mapsto q)$
- $(p \mapsto q) \wedge (q \mapsto r) \rightarrow (p \mapsto r)$
- If p ever becomes true, then eventually q becomes true, but
- p need not remain true until q becomes true, and
- p may become true when q is already true to satisfy the condition.

As stated previously, the temporal logic provides the capability to theorize and prove correctness. In addition, a designer can gain insight into appropriate design techniques to specify desired behavior as it applies to fixed points, invariance, and progress. Although the temporal logic and predicate calculus contribute to the formality of the language and design process (by providing an axiomatic and denotational semantic), the lack of operational semantics limits the applicability of the language for the purposes of this thesis.

UNITY combines first-order predicate calculus and temporal logic to provide a syntax for extracting the maximum parallelism from an algorithm. The purpose of UNITY is to form a foundation for parallel programming, which implies that it provides the smallest set of concepts

needed for formally describing parallel programs. Unfortunately, UNITY is not executable due to properties of infinite execution of statements and lack of control flow.

5.2.7 Program Composition Notation (PCN). Program Composition Notation (PCN) is a parallel design environment for developing and executing parallel programs (44, 24). The PCN system was developed as an effort for the National Science Foundation, the Air Force Office of Scientific Research, and other Department of Defense agencies. Personnel from the Argonne National Laboratory, California Institute of Technology, and Aerospace Corporation provided the research support in developing the PCN environment.

One of the developers of UNITY, Chandy, was also involved in the development of PCN. Because of Chandy's involvement, the similarities between these languages is readily apparent. The PCN language formalizes several of UNITY's properties and assumptions into an operational semantic. Specifically defined control structures were added to maintain certain assumptions from the UNITY language. For example, UNITY assumes that the execution of multiple statements accessing a shared variable could not occur. The UNITY definition assumes that one of the statements executing would *block*, interleaving the statement execution. PCN formalizes this in its operational semantic for the *definitional variables* (described in a later paragraph) adding control structures to ensure the proper execution behavior.

The environment provides both a parallel programming language and a parallel programming system for software development. The environment focuses on the composition of tasks into a formal task system specification. The environment includes a sophisticated toolkit including: a compiler, linker, foreign language interface, standard libraries, process mapping tools, programmable transformation system, symbolic debugger, execution profiler, and trace analyzer. The toolkit facilities are machine-independent but are combined with a machine-dependent run-time system for complete design support. A description of the benefits of the various toolkit components is described further below.

The focus on composition, as opposed to the individual components (blocks, procedures, modules, tasks), separates the PCN method of development from other design methods. The core composition operators provided by PCN include parallel, sequential, and choice. The design may include additional user-defined composition operators through the use of the sophisticated toolkit. The ability to add the user-defined operators allows for a domain-oriented design environment; however, this evaluation will focus only on the core operators. In addition, the language provides two types of variables: conventional or *mutable* variables, and single-assignment or *definitional* variables. A description of the distinction between these variable types is provided below.

The syntax of the language borrows several features from the programming language C. Specifically, no assumptions are made about the syntax used to define the blocks; the compositions ignore their contents. This allows for the composition of FORTRAN or C code modules without interpreting their syntax and semantics for correctness. Although this provides a powerful feature for an implementation environment, this composition lacks the formality envisioned for the parallel software design environment proposed in Chapter 3.

A key to understanding the PCN environment is an understanding of the concurrent model on which it is based. PCN encompasses the following principles of parallel programming design (44):

- First-Class Concurrency: The designer should identify concurrency at the lowest level possible and not append it to a sequential model. Through the low-level identification of concurrency, maximum parallelization can be achieved.
- Controlled Nondeterminism: Except when explicitly identified otherwise, computation should be a function of a procedure's inputs. This feature excludes factors of programming through side effects. In addition, the guarded command and choice operator can be used to control nondeterministic features.

- Compositionality: The structure created by the composition of individual components should be easily understood. By relying on the recursive, hierarchical approach, larger components can be constructed from smaller primitive structures. Both the large programs and the individual components should be independently understandable.

- Mapping Independence: The expected computed result should be independent of the mapping to the parallel architecture; thus, effectiveness and correctness are independent of the mapping problem. The independence allows for multiple mappings to be considered for efficiency, scalability, and other performance criteria.

These four principles are embodied within PCN's concepts, constructs, and operators. For example, *definitional variables* provide an abstract, machine-independent model of both communication and synchronization. *Concurrent composition* is the fundamental mechanism used to construct complex programs from simpler components. *Nondeterministic choice* is used to specify nondeterministic actions when required. *Encapsulation of state changes* allows state change to be integrated into concurrent computations without compromising deterministic execution.

For the specification of concurrent applications, PCN includes the following features (44):

- *Definitional Variables:* Concurrent processes communicate and synchronize through the use of *definitional variables.* The concept of the definitional variable includes assumptions that preclude the concurrency issues of data coherency. Specifically, the definitional variable can be defined only once, through a single assignment. Before its assignment it contains the value *undefined.* A concurrent process accessing an undefined definitional variable is assumed to block until the variable is defined, ensuring correctness in execution. Specific variations to the definitional variables, through the use of higher level concepts of *streams*, allow for the practical use of these variables in an intensive communication environment. The stream provides a specific control structure and sequence for the passing of information.

- *Concurrent or Parallel Composition:* Concurrent composition allows the designer to construct a concurrent software architecture where tasks can execute in parallel. The designer groups the blocks within delimiters and uses a || to specify that the blocks can execute in parallel. The correctness of execution of the parallel implementation is assumed, determined by the designer's analysis.

- *Choice Composition:* Choice composition allows for selection between alternative program traces. Each block is preceded by a guard. The guard is one or more predicate test. The structuring of the choice operation is similar to that defined by Dijkstra's guarded commands (35). The construction of choices can be specified in such a way as to provide nondeterminism and synchronization. Because one or more guards can be true simultaneously, nondeterministic execution is introduced. Nondeterminism is an essential property of concurrent specification (23). Synchronization can occur through the use of choice composition and use of the appropriate guards. If a guard's tests cannot be evaluated because of a currently undefined variable, execution *suspends* until the variable is defined. This feature can be used to synchronize specific tasks through the use of definitional variables and data transfer.

In addition, PCN provides the following more traditional features:

- *Mutable Variables:* Mutable variables are used for the program variables. Initially, they contain an arbitrary value, but they can be assigned and changed throughout process execution. The primitive types for this variable include *character*, *integer*, or *double* (double precision floating point).

- *Sequential Composition:* Sequential composition is a method of combining program blocks in a sequence. The control flow follows the specific ordering and therefore execution is sequential. The designer must use sequential composition when a dependency exists between the blocks (program encapsulations).

Another feature of PCN's syntax that should be mentioned is the specialized pattern matching guard, ?=. The ?= construct is used not only to verify a pattern match between a variable and a list structure (returning a boolean value) but also to assign items of the list to specific variables. For example: Ls ?= $[x|Ls1]$ assigns the first item of the list to x and the rest of the list to $Ls1$; the construct also ensures that the list has at least one element returning a boolean value. By using this construct within a guard, both a variable assignment and a pattern match are achieved. The pattern match can additionally be used for the purpose of specifying transformations.

The PCN environment provides many powerful features for the specification of parallel programs. The language highlights the importance of the parallel composition view of software design and abstracts the synchronization and communication into a specific controlled variable (definitional variable). The environment lacks the breadth of high-level data structures and operations, and mathematical operators offered by REFINE, GIST, and CIP-L. The environment includes the capability to add transformations; however, further research is needed to generate and define the transformations and a specific transformation methodology.

5.2.8 Chapter Summary. The following paragraphs summarize each of the languages:

Specifications in REFINE are created for validating formal behavior, not efficiency. The wide-spectrum nature of the language allows for incremental builds and supports transformations for stepwise refinement of specifications. Unfortunately, REFINE does not adequately support concurrency; however, the REFINE environment does provide a method of extending and altering the language. The schemas defined for CIP-L, UNITY, or PCN could be codified using the DIALECT tool. The REFINE toolset also lacks specific mapping tools required for a parallel software development environment.

The structured approach to specification and design, along with a mathematical logic basis, make CIP-L an attractive prospect for inclusion in the proposed platform. The CIP project laid the foundation for many of the current views on transformational programming, and CIP-L provides a

mathematical logic for defining transformational mappings (94). In addition, the schema definitions provided in the CIP language description describe the required control structures for the addition of a new type of statement, the *parallel statement.*

The foundations of the GIST language development environment, as identified in (10), define the requirements for an automated programming environment. These requirements apply directly to the research for this thesis. The language environment was formulated around the use of a database, including specific constructs within the language for manipulating and accessing the database. The sophisticated database was the forerunner of the knowledge bases used in today's systems (10).

The CSP language has been extensively studied and used as a metric for other languages and models (7, 23, 57, 103, 114). The concepts of traces and streams are used for describing communications in many of today's parallel languages. The interleaving of events through time offers a pseudo-concurrency representation that enables logical proofs to be performed using finite sequences. The choice operation is equivalent to the choice used in PCN composition. An added attraction of the language is its ease at being transformed into a graphical model. The graphical model provides a software architectural view equivalent to a state-machine description. However, pure CSP does not provide an execution capability.

UNITY provides a new perspective for parallel program specification. Chandy and Misra's research into the temporal logic proofs applied to parallel software specification contributed to this new perspective. The formal denotational semantics of the language combined with the temporal logic proof structure (axiomatic semantics) offers a powerful environment for understanding more about concurrent properties and concurrent applications. However, UNITY includes properties and assumptions (such as lack of control flow and infinite statement execution) which do not map easily to an operational semantic, if at all. Because of the complexity of creating the operational

semantics, UNITY would be a difficult language to include in an automated parallel development environment.

The PCN environment provides the best platform for the specification of concurrency. PCN's composition constructs allow for concurrency specification from the statement to the procedural level. The tools of the environment allow for the incorporation of domain-specific constructs, operations, and transformations through language extensions. The mapping tools allow for optimization of the design for a specific implementation. Unfortunately, the formalism of PCN focuses purely on the composition constructs. The PCN language could be extended to add wide-spectrum language features. This would require the addition of a schema language and extension to the PCN compiler. The current PCN documentation (44) provides only a limited description of the language extension feature, explaining that extensions are possible but not describing how to design them.

PVM was very different from the other specification environments and development platforms. PVM provides an abstraction of the control structure required to synchronize and communicate between processes. The *virtual machine* view reduces complexity by allowing the designer to design a machine-independent problem specification. The concept of separating the specific control structure from the problem specification was also used in UNITY.

This chapter provided a description of several specification languages and environments. The descriptions included the major concepts behind each language's syntax, semantics, and concurrency. Each of the languages have specific features that can be incorporated into a conceptual design of the parallel software development environment described in Chapter 3. The next chapter describes the language requirements for the parallel software development platform. It then analyzes and compares these languages with the requirements.

VI. Requirements for Parallel Language and Environment

6.1 Introduction

A conceptual environment for the semi-automated development of parallel software was introduced in Chapter 3. The proposed design environment was intended to encompass all aspects of parallel software design, from the formal specification of an application solution to its implementation on a specific parallel system. In order to facilitate a better understanding of the requirements for such a conceptual environment, the previous chapters have evaluated various aspects of parallel software design.

Of particular interest to the scope of this initial research is the explicit parallel language constructs. The transformation characteristics of the proposed environment are dependent upon a wide-spectrum *parallel* language capability. In addition, various other requirements can now be identified for the language and the semi-automated environment.

This chapter defines the initial requirements of a parallel programming environment for the application of concurrent features, formal methods, domain-oriented design, and software architecture modeling techniques. Several languages and environments were evaluated based on their formality (formal syntax and semantics) and concurrency. These form a subset of the overall requirements for the parallel software development environment.

This chapter summarizes the discussion of the previous chapters and synthesizes the evaluation into explicit requirements for the conceptual design of the proposed environment. It is not the intent of this list to represent a complete list of the requirements. Its purpose is to narrow the choice of possible candidate specification languages for a feasible implementation of the environment. The formal languages and environments are then evaluated based on these requirements.

Section 6.2 provides a summary of the information presented in Chapters 4 and 5. The analysis of this information is then incorporated into an informal list of requirements. Section 6.3 evaluates the specification environments described in Chapter 5. The analysis and evaluation

are summarized in Section 6.4. Unfortunately, none of the environments completely satisfy the requirements for the proposed parallel design environment; any of the candidate environments would require modifications or extensions.

Section 6.5 narrows the candidate specification languages to those with a realistic capability to be modified within the scope of this research. This section provides an informal discussion of the required modifications for the selection of a possible candidate and outlines the further analysis required. Finally, Section 6.6 summarizes the information presented in this chapter.

6.2 Requirements of the Language and Environment

We have now evaluated the use of models and modeling techniques, general automated programming techniques and concepts, the issues of parallel program design (including issues of hardware architectures, algorithm design, and informal techniques for task management), and current formal specification languages and environments. This section describes how these concepts can be applied to a formalized environment for parallel software architecture design.

This section identifies the requirements of a parallel specification language and support tools for use in our software design model. We must first understand the specific requirements for the language used within our conceptual platform. Two distinct stages in the development process, *formalization* and *optimization*, can be identified. Each phase requires specific transformations within the language. The domain knowledge and the composition of that knowledge into a specific software architecture must be formalized using the language. Transforming our specification into an efficient algorithm and mapping that algorithm to a target computer also places requirements on the language. There are several issues to keep in mind as we consider language requirements:

- The language is required to represent our domain and programming knowledge in the knowledge base. The knowledge should be structured for uniform access. The knowledge should

not include redundant information and should be organized to limit the amount of memory space required.

- Within our domain model, the static objects or entities and their dynamic behavior or activities need to be modelled. The software development process requires us to model the relationships between the objects and to constrain the objects through assertions. The domain model should capture the concurrent properties of the application.
- Explicit parallel language constructs are required to represent concurrency. The constructs need to support parallel, sequential, and choice composition. Additionally, the language should be extendible to provide the capability to create domain-specific composition operations.
- Besides the composition constructs, other high-level data types and operations are required. To ensure an appropriate power of expression, the language should include constructs for assignment, composites, conditionals, and iteration to provide sequence, iteration, and decision. These constructs are required to be sufficient to specify each partial recursive function. The inclusion of first-order predicate calculus is also desired. In addition, denotational semantics should provide the capability to create new data structures from the existing object types.
- A control structure for the composition of the domain knowledge and software architecture description into a specific instance of an application model (with concurrent compositions) is required. This composition system defines the specific interactions of the domain-specific knowledge and the programming knowledge.
- The composition of a software architecture should be constrained by formally defined rules to maintain consistency with domain requirements. The formalized rules apply domain-specific knowledge to constrain the application. The formalized rules may additionally be applied to other aspects of the design.

- A capability to represent formalized transformations is required. The transformations should be able to be applied incrementally and should be correctness-preserving within the language. A wide-spectrum capability could be helpful. In order to be able to make our assertions about consistency and completeness, as well as validated behavior, the language requires formally defined semantics.
- An operational semantic is required in order to create an executable specification. The execution of the specification can validate the composed software architecture's behavior. The operational semantic should be generalized enough to allow portability to many different architectures.
- The constructs within the language should facilitate interaction with the support tools, for example, accessing the knowledge base, using the parser and compiler, and mapping to a target hardware architecture.

6.3 Analysis of Parallel Languages and Environments

The various environments take different approaches to the specification of a problem: using different languages, making different assumptions to constrain the problem, and making different assumptions to represent implicit knowledge. The languages differ in their level of formality through inclusion of formal mathematical constructs. They also differ in their inclusion of various properties required for concurrent representation. Finally, each language includes different capabilities based upon their axiomatic, denotational, or operational semantics (56). This is important to our proposed environment because without an operational semantic the language is not executable.

This section evaluates the properties of the various languages and environments. Specifically evaluated are: the language's formality (formal syntax and semantics), the inclusion of concurrency properties, the inclusion of domain-specific and software architecture concepts, and the available tools of the environments and their language construct support.

6.3.1 Formalness of the Languages. The importance of a language's formal semantics is described in Section 2.3. Specifically, the section described operational, denotational, and axiomatic semantics and their characterization and contributions to describing the application and the domain.

The eventual language defined for the proposed environment requires a balance of these semantics. Within the transformational programming paradigm, the semantics must be consistent between transformations. The control structure that allows for the execution of a language must maintain the semantics in a program's execution.

The languages evaluated provide different levels of these formal semantics. Their formality is provided through their use of defined constructs from first-order predicate calculus, temporal logic, lambda calculus, algebraic types, and set theory, as well as their own formally defined additions. The following paragraphs address each of the languages considered.

The PVM constructs are dependent upon the machine-specific daemons. The daemons provide alternative operational semantics depending upon the machine of implementation. PVM merely provides a library of commands for interaction with the PVM control daemon. The library of routines are defined by the machine-specific compilers, possibly resulting in different behaviors on different machines. The PVM User's Guide provides the definition of these routines, but there is no proof of equivalence to ensure consistency between alternative machine representations. In order to use PVM with a specific machine, the machine requires a specific daemon to be loaded to allow for PVM's use.

UNITY and CSP provide their formalisms using denotational and axiomatic semantics. The languages do not provide an operational semantic to allow an execution capability. UNITY was specifically created to support parallel software development. UNITY defines specific program properties (such as fixed point and invariance) and uses temporal logic for consistency and correctness proofs of programs. When using UNITY, a designer uses properties of invariance, progress, and fixed-point in consistency, completeness, and correctness analysis. In addition, UNITY includes

formal syntax borrowed from first-order predicate calculus and the Pascal programming language. CSP, on the other hand, uses concepts of language theory and state-based theory to provide its formalisms. CSP's formal syntax is based upon user defined events, processes, and traces, and the functions applied to these. Although they were not analyzed in this research effort, future research should explore the variations of CSP (CCS and CCSP) that have been developed to provide an operational semantic.

Although PCN includes formal semantics for its composition constructs and variable definitions, it allows the statement blocks to include constructs of implementation languages. For example, a designer could compose blocks of C or FORTRAN code into an application. The C or FORTRAN code may be mapped to an implementation by separate machine-specific compilers. The PCN compiler ignores the contents of the blocks composed creating a formal semantic for the high-level program that assumes the correctness of the individual blocks. The C or FORTRAN code included in these blocks is compiled separately and requires separate formal semantics. For a formal proof, the correctness of the individual blocks would also need to be proven. PCN does not include the breadth of high-level data types and operations or the mathematical constructs that are provided in the other specification languages. PCN does not have a complete breadth of high-level data types within its syntax and uses specific constructs borrowed from the C programming language.

CIP-L, GIST, and REFINE were specifically designed for an execution capability; each provides an operational semantic. Each of languages includes the ability to extend the language by using the denotational semantics. Using the denotational semantics, new objects can be described based upon the primitive constructs of the language. Thus, the denotational semantics can be used in the creation of domain-specific terminology within the framework of the language. The language constructs borrow a great deal from logic theories, providing a consistent use of mathematical terms. The languages include a wide-spectrum language capability that is fundamental to

a transformation environment. In addition, a breadth of high-level data types and operations is available.

Each of the languages suffers from the high learning curve usually associated with formal languages (10, 56, 94). Specifications written in these languages may be difficult for human interpretation. A strong understanding of predicate calculus, algebra theory, and set theory is required to familiarize oneself with the language and specification interpretation. The power and flexibility of the language's denotational semantics and the appropriate specification organization can help to overcome this limitation. Of the languages evaluated, the REFINE and CIP-L languages provide the greatest flexibility through their denotational semantics.

6.3.2 Concurrency. The inclusion of concurrency in each of the various models is dependent upon each model's conception of time. Time is viewed as occurring as either a continuous or discrete event. In order to be specified and implemented on a computer, time is discretized. Temporal logic allows for representation of time as a continuum, where the temporal operators identify the discrete relationships between events. On the other hand, interval calculus interleaves discrete events over an interval. The alternative for concurrent representation in current languages is provided through adding temporal logic, interval calculus, primitive constructs for parallel composition, or constructs for specifying expected event sequences (7, 13, 23, 44, 57, 94). Additionally, the representation of concurrent properties can be provided through explicit constructs or through implicit properties of the language.

As we described in Chapter 4, a given implementation requires a specific control structure to maintain the consistency and correctness of the specification. The languages make certain assumptions and constraints for process interaction that impact the required control structure. Transformations to an implementation version are impacted by the operational semantics of the language. The languages also provide various levels of concurrent representation, from statement level to procedural level. The following paragraphs list these issues for the languages evaluated.

6.3.2.1 REFINE. REFINE does not include explicit constructs for composing parallel applications; however, REFINE does have some features that could be used if the language were extended to include concurrency. These include statement grouping and nondeterministic execution. The concurrent compositions could be added using REFINE's underlying schema language and the DIALECT tool.

6.3.2.2 UNITY. Although UNITY does not contain operational semantics, it is included in this analysis to highlight existing parallel constructs. UNITY avoids the issues of control structure until mappings to a specific implementation architecture are required. UNITY provides explicit constructs (|| and []) for composing groups of statements to allow concurrent execution. The sequential composition of statements is also provided by a comma (,), used for separating statements. In mapping to a hardware architecture, specific schemas are applied to transform these representations. However, UNITY does not provide an adequate formal definition and operational semantic for these schema mappings. For UNITY to add operational semantics, || would require a synchronized control structure for simultaneous execution, whereas [] should allow for asynchronous concurrent execution, where the statements can execute in a nondeterministic order.

6.3.2.3 PCN. The PCN language was designed explicitly for the parallel environment and therefore contains several features for the composition of parallel programs. Because Chandy was involved in the development of both UNITY and PCN, there are several similarities between these languages. The base constructs allow for nondeterministic choice, parallel, and sequential composition. These three operators are similar to those available in UNITY. Synchronization and interprocessor communication rely upon the semantic definition of the definitional variable. The restrictions placed upon these variables allow for data coherency but rely on the assumption that a processor blocks when these variables are undefined. The blocking of the processor relies on a specific control structure being added to ensure this property of definitional variables

is maintained. Thus, proper execution must ensure that the programming language and operating systems execution is consistent with the operational semantics.

6.3.2.4 GIST. The designer specifies explicit composition of concurrent tasks within GIST through an agent. The agent may include several concurrent actions, but these actions can not easily be mapped to a parallel architecture. This limitation is due to a lack of formal definition for the required control structure for the action's concurrent execution and therefore impacts the mapping of the individual actions to a parallel architecture. In addition, the specified demons rely on the support of the existing control structure. A specific interface for the demons and operating system needs to be identified, adding to the difficulty of transforming a specification into an implementation version. Concurrency is provided at a procedural level only.

6.3.2.5 CIP-L. The CIP programming environment appears to provide several features required for the envisioned platform. However, a complete description of the current CIP design environment could not be found. The specification of the parallel statement type and the definition of required control structures for maintaining the correctness of these statements was defined for the multiprocessor. These schemas would require further evaluation for the extension to distributed environments.

6.3.2.6 PVM. PVM provides concurrency through a mapping of multiple tasks to multiple processors. The parallel system is modeled as a virtual machine and thus the specifics of the communication and memory management control structures are abstracted. The PVM specification describes a specific parallel algorithm and its mapping to a virtual environment. The specific machine daemons interface with the actual environment to execute the required statements. The concurrency is therefore denoted through the specific mapping generated.

6.3.3 Software Architecture and Domain-Specific Knowledge. Each of the languages has the capability to structure the specification to accommodate concepts of domain knowledge

and software architecture. Fundamental to this capability is the language's ability to represent objects (static entities), object relationships (mapping and functions between objects), actions or activities (dynamic behavior of objects), and assertions and constraints on the objects. The language must provide for the abstraction and encapsulation of the domain-specific properties. The new encapsulated data structures (and their available operations) are named and referenced using a mapping to domain-specific terminology.

For use of a specific software architectural model that allows for concurrency, the properties of nondeterminism, task dependency, and time become critical to the model. A typical approach to state-based models is to represent time as a sequence of static snapshots, where *next state* is a property of interest. Each snapshot is a frame resulting from a specific modeled (defined) event. In this way, progression of the model's dynamic behavior provides a guarded transition. The task dependency is used to determine allowable parallelization. Nondeterminism allows for the interleaving of tasks for distributed concurrency. It is provided by allowing for multiple threads of control to initiate at a given point. Synchronization is represented through a specific dependency and the joining of the threads of control.

In the software architectures and modeling techniques evaluated (see Chapter 2), the dynamic behaviors are modeled through events, states, and state changes, together with a concept of time. The impact of these properties makes first-order predicate calculus inadequate due to the difficulty of time representation (55, 76). The software architecture model requires temporal logic or interval calculus principles to account for time.

6.3.4 Environment's Support Tools. Because of their execution capability, REFINE, CIP-L, GIST, and PCN provide an extensive array of automated support tools. Because an evaluation of these tools was not a major focus of this research, only a general evaluation of the PCN and REFINE tools was conducted. In addition, several support tools for mapping specific software

architectures to hardware platforms were evaluated. Specifically, the PYRROS, PARSA, and Task Grapher architecture mapping tools are described in Appendix E.

The functionality of the tools used in both PCN and REFINE environments are very similar. The PCN environment focuses on design for software optimization and parallel software implementation. This emphasis is apparent from the available toolset. PCN includes architecture mapping tools, trace analysis tools, transformation tools and an execution profiler for the support of parallel software design. On the other hand, the REFINE environment focuses on the formalization of the specification, initially ignoring the requirements for efficient execution. Because of the sequential nature of the current REFINE environment, no parallel support tools are included. However, REFINE provides tools for knowledge base (or object base) support, consistent with its overall emphasis on knowledge-based programming.

Obviously, the proposed environment requires extensive support through use of automated tools. The inclusion of tools that support domain-specific formalization, transformations for optimization, and mapping the software architecture to a specific parallel computer are required for the conceptual parallel software design environment described in Chapter 3. The existing tools for PCN, REFINE, and architecture mapping systems indicate that an adequate capability exists to meet the requirements of this environment using current technology.

6.4 Analysis Summary

REFINE and GIST includes a range of formal constructs and knowledge base tool support. The CIP project objectives imply a knowledge base capability (54, 94), but this could not be validated through specific documentation. PCN and PVM provide a limited *knowledge base* through their existing libraries. However, the term knowledge base is used loosely due to their limited capabilities for a sophisticated knowledge-based software engineering approach to design. Because

they are not executable languages, no specific knowledge base or support tools are provided for UNITY and CSP. Therefore, the best capabilities are provided by REFINE.

The specification execution capability requires the use of operational semantics. REFINE, CIP-L, GIST, PVM, and PCN provide operational semantics, but one must also consider: a) whether a complete set of constructs is available and b) whether the operational semantics for the current constructs support our transformational programming paradigm.

The transformation requirements strongly suggest the use of a wide-spectrum language incorporating formal and explicit parallel constructs. The languages GIST, REFINE, and CIP-L are all wide-spectrum languages, whereas the others are not. CIP-L and GIST were designed to include concurrent features, while REFINE can be extended to include concurrency in the same manner as CIP-L.

- UNITY and CSP are formal languages and support concurrency, but both lack an operational semantic. They are therefore not considered as candidates for implementation. However, both UNITY and CSP have an extensive proof process using temporal logic and language theory which can be applied to axiomatic semantic.
- PVM provides limited features for supporting a transformational programming environment. PVM lacks far too many of the required features for the proposed environment and is therefore eliminated as a candidate. However, PVM does provide a unique perspective on balancing the abstraction of the implementation environment and communication control structure in its operational semantic.
- CIP-L builds the concurrent features from the primitive kernel constructs of the language. The language documentation defines a set of concurrency features for parallel composition, nondeterministic choice, and protection for shared variables. CIP-L also has the support of an extensive transformational programming methodology through *morphisms* (94). It pro-

vides both a wide-spectrum capability (with operational semantics) and concurrency features. However, CIP-S (the CIP-L System environment) is not readily available.

- GIST provides an exceptional environment for knowledge-based software design. However, it can be argued that GIST provides limited concurrency. The *agents* provide only a procedural level of parallelism, which would limit the capability of specifying fine-grain parallelism and the mapping to a multiple processor architecture. GIST cannot be easily modified to add the required parallelism. Also, REFINE and CIP both included lessons learned from the design of GIST (and hence represents improvements over GIST); thus it is removed from further consideration.

- PCN offers the most extensive support of concurrency and specification optimization. However, PCN includes limited support for formalization of the application. There are also limited high-level data structures and mathematical operators. No capability for transformations through a wide-spectrum language is specifically included; however, the PCN manual does identify a transformation tool and the ability to extend the language to add domain-specific operators. One could also evaluate the addition of

- REFINE has the most extensive support for a knowledge-based software environment. The Software Refinery environment was designed specifically for knowledge-based software development and the object-base support allows for an extensive transformation capability. Its wide-spectrum features, the availability of an extensive assortment of high-level data structures, the object-base support, and explicit mathematical operators make Software Refinery an ideal environment for transformational programming. Unfortunately, REFINE's operational semantics do not include concurrency. However, REFINE's extension capabilities could be used to add specific parallel constructs. The design of extensions would require further research but could incorporate features from other parallel specification languages (such as PCN, UNITY, and CIP-L).

6.5 Detailed Language Requirements

Through the requirements analysis, the possible candidates for the conceptual design are reduced to PCN and REFINE. Due to the time limitations for the research, only one alternative could be evaluated. Therefore, the REFINE specification and programming environment is used as a basis for the conceptual design of the proposed design platform described in Chapter 3. The determining factors were:

- Although PCN had the most extensive set of parallel language constructs, it was not a wide-spectrum transformational environment. A wide-spectrum capability and knowledge base support would be very difficult to add to PCN. PCN does not include schema language support, extensive high-level data structures and operations, or explicit transformation constructs.
- REFINE's denotational semantics allow for the creation and use of domain-specific objects and grammars. The information can be stored and manipulated in the object base, which can be used to create a sophisticate knowledge base support environment. Although REFINE does not currently include parallel constructs, it provides a capability for extension. One can use the DIALECT tool to extend REFINE's capabilities with parallel composition constructs. We could use the knowledge gained from the evaluation of the other languages to define a complete set of parallel operators for REFINE.

In evaluating the possible parallel constructs to add to REFINE, we can gain a better understanding of the required constructs by evaluating the concurrency features of the other languages evaluated. Their concurrency features are summarized below.

- PCN provides specific constructs for parallel composition (||) and nondeterministic choice (?) applied to a set of statement blocks. The ***definitional variables*** are used to define shared vari-

ables within PCN. No features are available for specifying parallel enumeration or iteration; the design must explicitly decompose these features through the use of recursion.

- Parallel composition within UNITY provides two alternatives: asynchronous, nondeterministic parallel composition ([]) and synchronous, deterministic parallel composition (||), also applied to a set of statement blocks. UNITY uses specific assumptions for shared variables that are transformed to a data type of a *channel* in mapping to a distributed architecture. Nondeterministic choice is embodied in the [] construct combined with various methods of statement *guards*.

- CSP defines two alternatives for concurrency. The first assumes that the processes share an event (thus synchronized) and views this as a single event: the second assumes that the processes are independent and can be interleaved (thus asynchronous and nondeterministic). Processes separated by | represent *choice* within CSP.

- CIP-L specifically defines a new algebraic type for a parallel statement and shared variable. Constraints are then defined and applied to these new types. The constraints create a specific control structure for protecting shared variables and preventing statement conflicts. A parallel composition is defined for combining statements for concurrent execution. Nondeterministic choice is provided through a defined structures based on Dijkstra's guarded commands.

One can easily recognize the similarities between the different language representations. The spectrum of sequencing ranges from a strict serial execution to strict synchronized concurrent execution. The spectrum in between allows for nondeterministic interleaving of tasks. The full spectrum can be represented through a parallel composition operator, a serial composition operator, a nondeterministic choice operator, and a synchronization control variable. We can use these features in defining the REFINE language extensions.

The extension of REFINE creates a new language environment which we refer to as P-REFINE. As stated above, REFINE would need to be extended to include explicit parallel op-

erators. To add concurrency to REFINE execution, the operational semantics for the operators would need to be defined and a specific execution controller created. The parallel composition operators would require integration into the current REFINE environment through these operational semantics.

Constructs for parallel composition and nondeterministic choice may be added to REFINE by modifying its syntax. The definitions of REFINE's language constructs are provided in a separate package that is imported at compile time. The package can be extended to include the new parallel constructs for P-REFINE. The operators would also require an explicit protected control variable for communication and synchronization, as defined for CIP-L and PCN. The properties of the composition operators would also need to allow transformations and mappings to a parallel computer environment. The support toolkit for REFINE would need to be extended to include these optimization and mapping tools.

The difficulty in defining and integrating the parallel constructs is in determining their required operational semantic. The operational semantic must include adequate control structures not only for the sequential execution, but also for any number of possible mappings to both distributed and shared memory parallel architectures. As shown in Appendix B, these architectures have alternative requirements for their operating systems and their defined interprocessor communications. This would require a different transformation process for mapping to the alternative architectures.

Borrowing from PCN, one possible semantic would be to allow for nondeterministic execution. The possible parallel executable statements are maintained on a queue to simulate execution. The specific control variables determine the synchronization and execution flow.

The difficulty of this approach is with operations requiring a strict synchronization. For example, the multiple assignment and synchronous parallel composition must occur as an atomic system action. Although atomic action has been defined and designed for sequential systems and

shared memory systems, it is difficult to apply to distributed architectures. Distributed systems may have the control features distributed and also distributed shared variables. The distributed system's operational semantic for the constructs would require a distribution of the control structure.

As another example, for the asynchronous parallel composition one must verify correctness of the composed statements using a nondeterministic ordering. This would require a validation of the all possible trace sequences. If the asynchronous composed tasks were 1,2, and 3; the composition would be required to establish that the same functional results exist for the full spectrum of traces (from (1||2||3) to (1;2;3)) and all combinations of nondeterministic interleaving in between. In other words, the asynchronous execution must allow for nondeterministic execution within a specified barrier. Therefore, the composition operator is required to be both associative and commutative. The task of defining the operational semantic to maintain these properties would be complex because any shared variables between the statements would impact the compositions and their potential mappings. Thus, restriction would be required to ensure the individual blocks within a composition are not dependent upon each other.

A complete set of parallel constructs requires the definition of appropriate control variables for synchronous and asynchronous communication. One possibility for the definition of these control variables would be to apply the approach taken in PCN. PCN's *definitional variables* are a distinct construct that, when combined with PCN's sequence, offer an effective representation of communication. In order to maintain the appropriate correctness, PCN's compiler performs specific syntax and semantic checks for this variable type. Additional dependency analysis algorithms validate that the application of these variables meet the formal constraints. The compiler applies specific control structures and operating system interfaces for their implementation.

A specific type, called a *block*, is used in the definition of the parallel composition and nondeterministic choice operators. The block structure is merely a grouping of statements that have been composed using either the serial, parallel, or nondeterministic choice operators. As described

above, the application of a parallel composition to a set of blocks would require that no dependencies existed between the individual blocks within the composition (no shared variables). Constraints would be added to provide a control structure to ensure specific concurrency properties were maintained for both blocks and shared variables used by the blocks. The synchronization and execution flow would be dependent upon the shared variables and their assignment. For a distributed system mapping, the shared variables would be transformed to an alternative *channel type* with any additional constraints provided.

The parallel operators may require a specific statement type, *parstat*, for defining the blocks. The application of a parallel composition to a block of statements would transform the statements into the *parstat* objects. Constraints would be added to provide a control structure to ensure that specific concurrency properties were maintained for parallel statements and any shared variables. The synchronization would be defined using the shared variables. For a distributed system mapping, the shared variables would be transformed to an alternative *channel type* to provide any additional constraints.

6.5.1 Expanding Support Tools. The requirements for the proposed toolkit for the parallel environment is based on the available support tools in PCN and REFINE specification environments, and other support tools used in parallel design (see Appendix E). The extensive definition of the required toolkit is the subject of future research. The toolkit might include: a compiler, linker, foreign language interface, standard libraries, process mapping tools, programmable transformation system, symbolic debugger, execution profiler, and trace analyzer. These tools should be augmented to allow the addition of domain-specific transformations and rules to assist in visualizing, prettyprinting, extending, and manipulating the knowledge base. In addition, parallel visualization tools with similar capabilities to the AAARF for showing system performance should be added (25).

The integration of the parallel mapping tools to the REFINE programming environment would require additional research. In addition, the existing REFINE tools would need to be evaluated for the integration of the proposed parallel language extensions.

6.6 Chapter Summary

This chapter described the requirements for the environment and language and proposed a design based on the existing support tools, languages, and design methods. This chapter informally discussed the required language constructs for specifying parallel applications and supporting a transformational programming process. We related the informal requirements to a possible implementation. The formal specification languages and environments were then evaluated against these requirements, and the REFINE and PCN specification languages were selected as the best candidates for additional research. The conceptual design proposes an extension to the REFINE language using a modified PCN syntax. REFINE is therefore used as a basis for the discussion of the proposed design. The proposed parallel extensions to REFINE, however, are based on the PCN composition operators. We attempt in the next chapter to combine the different characteristics of each of these environments into a new parallel specification and design environment for a conceptual design of the platform proposed in Chapter 3.

VII. Formal Definition of a Complete Set of Parallel Constructs

7.1 Introduction

The previous chapters informally characterized the parallel software design issues. These issues were then used to specify the requirements for the conceptual design of a parallel software development environment, emphasizing the language requirements for parallel software design. This chapter establishes a formal definition of the parallel constructs required for defining a complete set of parallel operators. The definition of the parallel constructs is based on a conceptual design of parallel extensions to the REFINE specification environment. The chapter presents the following aspects of the formal constructs:

- Section 7.2 provides an overview of the chapter organization and a general description of the basis of the formal proof structure.
- Section 7.3 defines the formal syntax for the parallel constructs.
- Section 7.4 explains the integration of the defined parallel constructs with high-level data types and operations of the REFINE specification environment. In addition, the transformation of the parallel constructs is briefly addressed.
- Section 7.5 defines the specific mathematical properties of the parallel operators, with examples of their use.
- Section 7.6 describes a formal proof of the consistency and completeness of the parallel operators for defining parallel applications.
- Section 7.7 describes and defines the formal semantics of the parallel operators.

7.2 Overview of Formal Proof Process

In ensuring that we have identified a *complete* set of operators for the parallel environment, we must characterize the formal definition of the parallel operators. In discussing the formal

properties of the parallel language, we must first characterize the formalization of a language and its definitions. The following paragraphs explain specific issues of formally defining *consistency* and *completeness* for the parallel language definition.

In (52), Goldblatt discusses axiomatizing the logic of computer programming through the definition of the external and internal logic using formal mathematical theories. The *external logic* of the computer program characterizes the software engineer's reasoning based on the program behavior, whereas the *internal logic* characterizes the operations performed by the computer in evaluating the boolean expressions (52). Golblatt creates a framework for formal proof theory and axiomatization specifically using characterization of the internal and external logic, as well as modal logic.

The software engineer may use the external logic to constrain the internal logic with the appropriate use of the operators, and by using a formal program derivation process. These issues raise the importance of using formal methods in the design process for ensuring the program design does not lead to an undefined (or non-terminating) result. The *consistency* of the two-valued logic system ensures that the use of the constructs does not lead to a contradiction.

An assertion of *completeness* of the parallel constructs relies on Church's Thesis for language theory (84). In (33), Dijkstra asserts that a formal system must include operators for assignments, composites, conditionals, and iterations (i.e., sequence, iteration, and decision) to specify a program to compute each partial recursive function. If this is a sufficient set of operators, then by Church's Thesis, a language with these constructs contains the capability to specify programs for all possible algorithms (52). In defining a complete set of parallel composition operators, we expand on the concept of sequence, iteration, and decision to incorporate concurrency. As stated previously, nondeterminism is a key component to concurrent specification. We contend that a complete set of parallel operators is created by combining the concepts of sequence, iteration, and decision with concepts of nondeterminism and concurrency, and the addition of nondeterministic choice.

The definition of the operational and axiomatic semantics requires the use of a specific formal system that can represent concurrency. The concurrency representation chosen uses a nondeterministic interleaving of events, or *pseudo-concurrent* representation. The approach is similar to the formal system used in (16), where the formal semantics for parallel operation are characterized through a state-transition system combined with nondeterministic interleaving of events. Other researchers have also applied parallel proof systems using state-transition models and nondeterminism; in fact, that is the concept applied in the CSP language (23, 24, 34, 59).

The specified operators have been defined to be consistent with the composition operators of PCN. PCN's composition operators were selected because their operators adhered to our objectives for expanding sequence, iteration, and decision. The PCN operational and axiomatic semantics is therefore used as a starting point and modified where required. PCN's operational and axiomatic semantics, as defined in (24), apply predicate calculus and a state-transition model in their definition.

The axiomatic semantic definitions include predicates that resemble the specific operators of temporal logic in order to incorporate a concept of time. The axiomatic semantics specify a formal design process using specific properties of fixed point, invariance, and progress. The operational semantics define the execution of the operators as they relate to a specific transition system. As stated previously, the transformations must establish a relationship between the axiomatic semantics and the operational semantics. The transition system model provides a formal system for characterizing this relationship (16).

7.3 Formal Syntax for Parallel Operators and Parallel Control Variable

Based on our previous assertions, a complete set of constructs can be created by expanding the sequential constructs of sequence, iteration, and decision. The expansion applies concepts of concurrency and nondeterminism to form the parallel constructs. The complete set of composition

operators proposed are: parallel composition, parallel composition with enumeration, sequential composition, nondeterministic choice, and a specific control variable for communication and synchronization. The formal syntax for these constructs are as follows (24):

Parallel Composition:

{ || | (local variables): $Block_1, Block_2, ...Block_n$ }

- The parallel composition initiates the execution of all of the blocks within its delimiters.
- Each block's control variables determine the synchronization between the set of blocks (e.g., A shared variable x may be included in each of the blocks generated by a process outside the scope of this composition. The defining of the shared control variable determines the block synchronization and execution order).
- No dependencies can exist between blocks (i.e., $Block_1$ cannot assign a value for a variable required for $Block_2$, ..., $Block_n$).
- The local variables define specific variables that are available only within the scope of the specific composition.
- Each block has the possibility to be mapped to an alternate processor.

Parallel Composition (with enumeration):

{ [] | (local variables): (enumeration range) | *Block* }

- This specific syntax is a special case of the parallel composition that allows for a more convenient notation. This construct allows for a specific iteration to be specified. Although this could also be accomplished by manually unrolling the statements into multiple blocks or by using recursion, this notation is simpler and facilitates a mapping to multiple processors not apparent in a recursive definition.
- The enumeration range provides a specific range for enumerating over a shared variable of the statement blocks. Multiple blocks are created using a different enumeration parameter. This may be viewed as a *loop unrolling*.
- The parallel composition initiates the execution of all of the blocks within its delimiters.
- Each block's control variables determine the synchronization between the set of blocks (e.g., A shared variable x may be included in each of the blocks generated by a process outside the scope of this composition. The defining of the shared control variable determines the block synchronization and execution order).
- No dependencies can exist between blocks (i.e., enumeration blocks).
- The local variables define specific variables that are available only within the scope of the specific composition.

- Each block has the possibility to be mapped to an alternate processor.

Sequential Composition:

{ ; | (local variables): $Block_1, Block_2, ...Block_n$ }

- The actions of each block must occur in sequence, i.e., $Block_1$ must complete before $Block_2$ is initiated.
- The local variables define specific variables that are available only within the scope of the specific composition.

Nondeterministic Choice Composition:

{ ? | (local variables): $Guard_1 \rightarrow Block_1, Guard_2 \rightarrow Block_2, ...Guard_n \rightarrow Block_n$ }

- The choices are nondeterministic due to the possibility of more than one guard condition being true at any given time, allowing the execution to progress in a nondeterministic manner.
- There is an inherent concurrency in this operation through its nondeterminism. Each statement group may be mapped to a different processor.
- The local variables define specific variables that are available only within the scope of the specific composition.

Communication Variables:

variable_name: comvar, type

(where type is integer, set, string, character, etc...)

- Using the keyword *comvar* invokes constraints for shared variable protection.
- The variable can be combined to form a special data type using the sequence construct to create a *channel* (used in mapping to the distributed environment).
- The variable is initially undefined and is only assigned a value once. A process attempting to access an undefined variable will block execution until assignment.

7.4 Integrating Parallel Composition with High-Level Data Types and Operations

The parallel constructs require a specific integration with high-level data types and operations. The integration relies on the definition of the *block structure*. REFINE blocks are currently defined

using the atomic types of expression and statement. An expression reduces to a boolean value and a statement causes a specific state transition or assignment. The current REFINE *block* definition is based on sequential composition. The integration of the parallel composition operators requires a modification to the block definition to add the new types of blocks for parallel composition, parallel enumeration, and nondeterministic choice.

Updating the definition of the REFINE *block type* may require the definition of a new parallel statement and parallel expression. The parallel statements or expression types would be used to uniquely identify these types for the application of any special constraints to the execution controller. The special control variables would also invoke specific constraints for their use in conjunction with the statements and expressions. However, the new types would still reduce to the specific atomic terms for statement and expression. Maintaining the statement and expression as the primary atomic terms is fundamental to the overall integration of the new terms.

For example, as stated previously, all current operations reduce to either a statement (e.g., invoking a function or variable assignment) or an expression (value comparison or predicate function). The internal logic of the computer ultimately reduces all well-formed formulas (wff - compound expressions and statements) to verify the syntax. Thus, the operational semantics are based on these definitions. Any major modifications to these definitions may require significant revisions to the entire environment. The defined operational semantics are based on the concepts of expression and statements and provide for the integration of the new block definition into the REFINE environment.

Each block, as used in the definitions for the new operators, can be a compound statement or expression. Each composition operator creates a block, thus allowing for a hierarchy of compositions. For example, one could apply parallel composition to a set of parallel composed blocks. The REFINE precedence constraints would require updating to include the new operators. The key to statement parsing would be the delimiters used to separately identify these compositions.

For the purposes of the formal syntax defined above, the delimiters were selected as a pair of curly brackets ({ }). One may prefer another alternative for the actual implementation due to the current special use of the brackets as a setformer operation. The designer of the extensions must weigh the trade-offs among alternatives.

The control variable requires a specific variable type creation. This new variable requires that specific protection constraints be placed on the variable. For example, a unique assignment statement for defining these variables would apply. The variable would contain two specific attributes: a boolean value determining if it is *defined* or *undefined*, and a value that can only be assigned if the variable is undefined. The assignment must occur within a protection scheme such as a monitor to ensure atomic action.

Applying this to the block definition, we can define the operation based upon the blocks. A block is either a definition statement (assignment of value to a control variable), an assignment statement (assignment of value to variable), a program call (execution of program statement), or a composed block (either a parallel composed block, choice composed block, or sequential composed block). The composed blocks must then be able to be reduced to either a parallel statement or a parallel expression interleaved in execution.

7.5 Mathematical Properties of Parallel Operators

The definition of the operators implies formal properties for the operators. Specifically, the associative, commutative, and reflexive properties for each operation should be identified. These properties could then be used in defining the possible transformations. The following section defines the specific mathematical properties of each of the operators and relates the operator to the specific transformation process.

From our previous analysis, we have determined that parallel and choice composition are associative and commutative, whereas sequential composition is only associative. For example:

- $A \| B = B \| A$, $A \| (B \| C) = (A \| B) \| C$
- $A ? B = B ? A$, $A ? (B ? C) = (A ? B) ? C$
- $A; (B; C) = (A; B); C$

Note: additional properties may apply depending on the REFINE implementation of the operational semantics. For example, one may choose to use the mathematical properties of associativity and commutativity to implement the parallel and choice composition as a series of binary relations. This would allow the use of specific set operations on the set pairs. One could then define specific mathematical properties based on the relation.

7.6 Completeness of Parallel Constructs

As discussed previously, by the work of Church and Dijkstra, a *complete* set of constructs for the serial environment is provided by sequence, iteration, and decision. REFINE's sequence, iteration, and decision constructs also allow for the specification of each partial recursive function and thus is complete. We have used REFINE for a starting point in defining our parallel operators. We have specifically applied nondeterminism and concurrency to each of these categories. Parallel composition is used to augment the serial sequencing with parallel sequencing, Parallel composition (with enumeration) allows for a parallel iteration, and nondeterministic choice allows for the possibility of parallel decision. The specific control variables are used to provide the synchronization and control execution flow, and are therefore required to define the execution flow.

A formal proof of this completeness is difficult and beyond the scope of this research. We therefore assume completeness in defining the operators.

7.7 Operational Semantics

This section defines the operational semantics for the parallel constructs for P-REFINE execution. The actual implementation requires a modification to the REFINE environment to create an *execution controller* to control the specific execution of the candidate statements for execution

and is beyond the scope of this initial research. The defined operational semantics are taken from PCN (24) and modified where required.

The operational semantics for the composition operators are defined using a state-transition system model. The transition system can be defined as follows (24):

A *transition system* is defined by

1. A nonempty set of *states*,
2. A subset of the set of states called the set of *initial* states,
3. A binary relation $\rightarrow$ (called a *transition*) between states, and
4. A *fairness* rule,

A final state may be defined as follows:

S is a final state $\equiv \neg\ (\exists T :: S \rightarrow T)$

Where S and T are states.

A computation is a specific finite trace of transitions leading to a final state or an infinite trace with *fairness*.

The specific states of the system are defined by the variable values. However, the special case of the control variable requires a specific definitional assignment and can only be assigned once. The control variables can be a specific expression value (called its *definition*) or a sequence of items (*channel*). The syntax for *definition* is defined as follows: x **def** e in S, where x **is defined as** e **in state** S, and *Undef.*x in S, where x **is undefined in state** S. The relation **def** is not symmetric. For example, it is possible that

(x **def** y) $\wedge\neg$ (y **def** x) be in a state.

The State of a Program:

The state of a program is given by:

1. A set V of variables, and a set C of control variables,
2. A value for each variable in V,
3. For each control variable v in C: (a) a boolean *Undef.v* and (b) a single definition for v if and only if $\neg$ *Undef.v* holds,
4. A set E *of points of execution* in the program; this is the candidate set of all statements initiated for execution.

Fairness applies to the selection from the candidate set, where, once a statement is selected, the system progresses to a state where it is not able to be selected again (no longer in the candidate set).

Each *block* requires a specific initiation and termination point. A *block* can be: a definition statement, an assignment statement, or a composed block (parallel-block, sequential-block, or choice-block).

State Transitions:

For a state transition $S \rightarrow S'$, let V, C, E be the set of variables, control variables, and candidate set in state S, respectively. Let V', C', E' be the corresponding sets in S'. Let $A = V \cup C$, and T' be the corresponding set in S'.

The predicate function *same* is introduced for convenience. *Same* is defined as follows: *same* $(v, S, S') \equiv$

$((v \in V \cup V') \wedge ((v = y \text{ in } S) \equiv (v = y \text{ in } S'))) \vee ((v \in C \cup C') \wedge ((v \textbf{ def } f \text{ in } S) \equiv (v \textbf{ def } f \text{ in } S')))$

Thus *same(v,S,S')* holds if and only if variable v has the same value or definition in both states S and S'.

The initiation and termination points for a block b are defined as $I.b$ and $T.b$, respectively.

The notation *[P]* represents "Predicate P holds for all states of the program." Using this to clarify **Undefined**, we have:

$[(\exists\ e :: x\ \mathbf{def}\ e) \equiv \neg(\mathit{Undef.x})]$ and

$[(x\ \mathbf{def}\ e) \Rightarrow (x = e)]$

The predicate *data.x* is defined as follows:

$[\mathit{data.x} \equiv (\exists z :$ z is a ground value or tuple: x *reduces_to* $z)]$

where reduction is a substitution of the defined values for the control variables in x.

Execution:

Execution is defined by the following possible transitions:

$S \rightarrow S' \equiv$
initiate_program_call(S, S') $\vee$
terminate_program_call(S, S') $\vee$
execute_definition_statement(S, S') $\vee$
execute_assignment_statement(S, S') $\vee$
initiate_parallel_block(S, S') $\vee$
within_parallel_block(S, S') $\vee$
terminate_parallel_block(S, S') $\vee$
initiate_sequential_block(S, S') $\vee$
within_sequential_block(S, S') $\vee$
terminate_sequential_block(S, S') $\vee$
initiate_choice_block(S, S') $\vee$
terminate_choice_block(S, S') $\vee$

Initiate_Program_Call:
There exist a call to a program (function, form, or rule) $f(x)$, where x is a sequence of parameters and

$$(V' = V \cup m) \wedge (C' = C \cup n) \wedge (\forall\, y : y \in n : (\mathit{Undef}.y \text{ in } S')) \wedge$$
$$(\forall a : a \in A : \mathbf{same}(a, S, S')) \wedge (I.f(x) \in E) \wedge (E' = E + I.b \text{ - } I.f(x))$$

where m is a set of local variables and n is a set of local control variables of f, and the body of f is a block b.

Terminate_Program_Call:
There exists an f, x, m, n as defined above, and

$$(V' = V - m) \wedge (C' = C - n) \wedge (\forall a : a \in A : \mathbf{same}(a, S, S')) \wedge$$
$$(T.b \in E) \wedge (E' = E + T.f(x) \text{ - } T.b)$$

Execute_Definition_Statement:
There exists a definition statement x = e where:

$$(M = M') \wedge (D = D) \wedge (\forall\, a : a \in A - x : \mathbf{same}(a, S, S')) \wedge$$
$$(I.(x = e) \in E) \wedge (E' = E + T.(x = e) - I.(x = e) \wedge (x \mathbf{def}\ e' \text{ in } S')$$

Execute_Assignment_Statement:
There exists an assignment statement $x \leftarrow e$ where:

$$(M = M') \wedge (D = D) \wedge (\forall\, a : a \in A - x : \mathbf{same}(a, S, S')) \wedge$$
$$(I.(x \leftarrow e) \in E) \wedge (E' = E + T.(x \leftarrow e) \text{ - } I.(x \leftarrow e)\ ) \wedge$$
$$(\forall y : (y \in C) \wedge (y \text{ appears in } e) : \mathit{data}.y \text{ in S}) \wedge (x = e' in S')$$

Where e' is obtained from e by substituting for each variable in e its value in S, and substituting for each control variable y in e the ground value to which y reduces in S.

Initiate_Parallel_Block:
There exists a block b where b is $\{ \parallel i : i \in K : a_i \}$:

$$(V' = V \cup m) \wedge (C' = C \cup n) \wedge (\forall\, y : y \in n : (\mathit{Undef}.y \text{ in } S')) \wedge$$
$$(\forall\, a : a \in A : \mathbf{same}(a, S, S')) \wedge$$
$$(I.b \in E) \wedge (E' = E \cup \{\ I.a_i \mid i \in K\ \} \text{ - } I.b)$$

Where n and m are sets of local variables.

Within_Parallel_Block:
There exists a block b where b is $\{ \parallel i : i \in K : a_i \}$:

$$(V = V') \wedge (C = C') \wedge (\forall\, a : a \in A : \textbf{same}(a, S, S')) \wedge$$
$$(\exists i : i \in K : (T.a_i \in E) \wedge$$
$$(\exists j : j \in K) \wedge (j \neq i) : \textit{executing}.a_j \in S) \wedge (E' = E - T.a_i))$$

Note, this allows certain blocks to terminate, while others continue.

Terminate_Parallel_Block:
There exists a block b where b is $\{ \parallel i : i \in K : a_i \}$:

$$(V' = V - m) \wedge (C' = C - n) \wedge (\forall a : a \in A : \textbf{same}(a, S, S')) \wedge$$
$$(\exists i : i \in K : (T.a_i \in E) \wedge$$
$$(\exists j : j \in K) \wedge (j \neq i) : \neq(\textit{executing}.a_j \in S)) \wedge (E' = E + T.b - T.a_i))$$

Where n and m are defined as above.

Initiate_Sequential_Block:
There exists a block b where b is $\{ ; i : 0 \leq i \leq n : a_i \}$:

$$(V' = V \cup m) \wedge (C' = C \cup n) \wedge (\forall\, y : y \in n : (\textit{Undef}.y \text{ in } S')) \wedge$$
$$(\forall\, a : a \in A : \textbf{same}(a, S, S')) \wedge$$
$$(I.b \in E) \wedge (E' = E + I.a_0 - I.b)$$

Where n and m are sets of local variables.

Within_Sequential_Block:
There exists a block b where b is $\{ : i : 0 \leq i \leq n : a_i \}$:

$$(V = V') \wedge (C = C') \wedge (\forall\, a : a \in A : \textbf{same}(a, S, S')) \wedge$$
$$(\exists i : 0 \leq i < n - 1 : (T.a_i \in E) \wedge (E' = E + I.a_{i+1} - T.a_i)$$

Terminate_Sequential_Block:
There exists a block b where b is $\{ ; i : 0 \leq i \leq n : a_i \}$:

$$(V' = V - m) \wedge (C' = C - n) \wedge (\forall a : a \in A : \textbf{same}(a, S, S')) \wedge$$
$$(T.a_{n-1} \in E) \wedge (E' = E + T.b - T.a_{n-1})$$

Where n and m are defined as above.

Initiate_Choice_Block:
There exists a block b where b is $\{ ? i : i \in K : g_i \rightarrow a_i \}$:

$$(V' = V \cup m) \wedge (C' = C \cup n) \wedge (\forall\, y : y \in n : (\textit{Undef}.y \text{ in } S')) \wedge$$
$$(\forall\, a : a \in A : \textbf{same}(a, S, S')) \wedge (I.b \in E) \wedge$$
$$(((\forall i : i \in K : g_i \textit{ reduces\_ to false}) \wedge (E' = E + T.b - I.b)) \vee$$
$$(\exists i : i \in K : (g_i \textit{ reduces\_to true}) \wedge (E' = E + I.a_i - I.b)))$$

Where n and m are sets of local variables.

Terminate_Choice_Block:
There exists a block b where b is $\{ ?\, i : i \in K : g_i \rightarrow a_i \}$:

$(V' = V - m) \wedge (C' = C - n) \wedge (\forall a : a \in A : \mathbf{same}(a, S, S')) \wedge$

$(\exists j : j \in K : (T.a_j \in E) \wedge (\mathrm{E'} = \mathrm{E} + T.b - T.a_j)$

Where n and m are defined as above.

7.8 Chapter Summary

In this chapter, we define new parallel operators for the REFINE specification language. The definition of these new operators is based upon the analysis provided in the previous chapters. The informal analysis of these previous chapters is used to provide a specific formal syntax and formal semantics for the defined operators.

We contend that the defined operators represent a complete set of parallel operators. The formal definition of the operators is related to previous work by Church and Dijkstra. The relationship to this previous work is used to support our contention, and does not represent a formal completeness proof of the constructs.

The next chapter demonstrates the use of the proposed operators in defining parallel applications. The specific applications of parallel sort and parallel search are defined using these operators.

VIII. Example Problems

8.1 Introduction

This chapter provides example problems, specifically, a parallel merge sort and a parallel best-first search, to demonstrate the use of the parallel constructs. The merge sort example is first specified using different specification languages (REFINE, UNITY, PCN). The example specifications demonstrate the similarities and differences of the languages. A general best-first search algorithm is then provided. We then revisit each of these algorithms using the proposed parallel constructs for P-REFINE, the sort example at a low level, the search example at a higher level.

8.2 A Parallel Sort Algorithm

The general sorting problem can be characterized by the following functional description:

Given an input list of objects, generate a list of objects that is a permutation of the input list in some specified order.

The various methods of parallelization rely on a divide-and-conquer (or recursive) approach to the problem decomposition. The problem can use either data decomposition, control decomposition, or a mixture of both. Sorting algorithms that use a divide-and-conquer approach include bitonic merge sort, quick sort, and odd-even sort. For the purposes of this research, the bitonic merge sort (as described in (101)) will be used. The choice of the bitonic merge sort was arbitrary; any of the other divide-and-conquer algorithms could have been selected. The algorithm demonstrates the use of explicit constructs for representing parallelism for a polynomial algorithm.

In 1968 Batcher introduced a parallel-sorting algorithm with time complexity $O(\log^2 n)$. The algorithm applies a SIMD model of design. The bitonic merge sort divides the list up into m sublists of equal size and sends one sublist to each processing element (PE). The PE then performs an initial sort on its sublist. The PE uses a bitonic algorithm to merge the sorted lists.

The fundamental operation for this algorithm is the compare-exchange: two numbers are routed into a comparator, where they are exchanged (if necessary) so that they are in the proper order.

A bitonic sequence is a sequence of numbers $a_0, a_1, \ldots, a_{n-1}$ with the property that

1. $\exists i, 0 \leq i \leq n-1$, such that $a_0 \ldots a_i$ is monotonically increasing and $a_i \ldots a_{n-1}$ is monotonically decreasing, or else
2. $\exists$ a *cyclic shift* of indices so that the first condition is satisfied.

A single compare-exchange step can split a single bitonic sequence into two bitonic sequences, each half as long as the original sequence. Thus a bitonic sequence of length 2^k can be sorted in k steps by recursively applying the same compare-exchange operation to the resulting bitonic sequences. If instead of a bitonic sequence, we start with an unsorted sequence, we apply the perfect shuffle (100:86). Thus we begin with bitonic sequences of length 2. Thereafter, at each step, the lengths of the bitonic pieces double (38).

The bitonic merge sort in (101) is presented for SIMD machines, so it does not include synchronization between nodes. Implementation using a MIMD machine requires additional synchronization. This synchronization is accomplished through the message passing on a distributed multicomputer. The node operating system blocks the process until the synchronization message arrives.

The bitonic merge sort is specified below using UNITY (23), PCN (44), and REFINE (103). The different descriptions are provided to demonstrate the different problem representations and different language constructs.

UNITY Merge Sort:

The following UNITY description of Bitonic merge sort algorithm was derived from the description by Quinn (101). The implementation of this algorithm uses host (controller) and node (PE) programs in a SIMD approach. The controller **sort** loads the PE program **Bitonic**, and partitions the data amongst the processors for sorting. Each node begins by sorting its portion of the data. The data is then exchanged and merged as described by the algorithm below. The algorithm can be described in UNITY as follows:

Program *Bitonic Merge Sort*

assign

$\langle\| \forall k : \ 0 \leq k \leq NP \ ::$ (NP = # processors)

$\langle\| \forall i : \ 0 \leq i \leq m-1 \ ::$

$\langle\| \forall j : \ 0 \leq j \leq i \ ::$

$b_k, a_k := max(a_{k+2^j}, a_k), min(a_{k+2^j}, a_k)$ if $(k \bmod 2^{j+1} < 2^j) \wedge (k \bmod 2^{i+1} < 2^i)$

$b_k, a_k := min(a_{k+2^j}, a_k), max(a_{k+2^j}, a_k)$ if $(k \bmod 2^{j+1} < 2^j) \wedge \neg(k \bmod 2^{j+1} < 2^j)$

$a_k := b_{k-2}$ if $\neg(k \bmod 2^{j+1} < 2^j)$

$\rangle$

$\rangle$

$\rangle$

end *Bitonic Merge Sort*

NP is the number of processors, and k corresponds to the kth processor. Each a_k corresponds to the elements or sub-lists to be sorted.

PCN Merge Sort:

```
MERGESORT(Ls,SLs)
{? Ls ?= [u,v|Ls1] ->
        {|| SPLIT(Ls,LLs,RLs),MERGESORT(LLs,LLs1), MERGESORT(RLs,RLS1),
            MERGE(LLs1,RLs1,SLs)
        }
    default -> SLs = Ls        % default is SIZE(Ls) = 0 or SIZE(Ls) = 1
}

SPLIT(Ls,LLs,RLs)
{? Ls ?= [u,v|Ls1] ->
        {|| LLs=[u|L1], RLs=[v|R1], SPLIT(Ls1,L1,R1)},
   default -> {|| LLs = Ls, RLs = []}
}

MERGE(Ls1,Ls2,Ls3)
{? Ls1 ?= [] -> Ls3 = Ls2,
   Ls2 ?= [] -> Ls3 = Ls1,
   Ls1 ?= [x|L1s], Ls2 ?= [y|L2s] ->
       {? x<=y -> {|| Ls3 = [x|L3s], MERGE(L1s,Ls2,L3s)},
          x>y  -> {|| Ls3 = [y|L3s], MERGE(Ls1,L2s,L3s)}
       }
}
```

The PCN specification uses a recursive procedural representation of the problem (24). *Ls* represents the input list and *SLs* is the sorted list. The PCN model uses a specific type of communication variable, the ***definitional variable***, to synchronize execution. The variable is initially undefined and then may be assigned a value only once. A program statement attempting to execute with an undefined variable will block until the variable is assigned. The operational view of this program can be seen as generating a queue of executable statements. The individual statements reside on the queue until they are able to execute. The recursive calls create a unique instance for the individual variable. The execution of the program performs $\log n$ ***splits*** and $\log n$ ***merges***.

REFINE Merge Sort:

```
function MERGESORT(Ls:seq(integer))
returns
(SLs: seq(integer) | BAG-EQUAL(SEQ-TO-BAG(Ls), SEQ-TO-BAG(SLs)) and LE-ORDERED(SLs))
% This ensures a permutation and the Least element first ordering
= if Ls = [] then []
  elseif rest(Ls) = [] then [first(Ls)]
  else let (var Y: tuple(seq(integer), seq(integer))
  = SPLIT(Ls))
    MERGE(MERGESORT(Y.1), MERGESORT(Y.2))


function SPLIT (x: seq(integer) | x~= [])
returns (p: tuple(seq(integer), seq(integer))
        | concat(p1,p2) = x
          & (size(p1) = size(x) div 2)
          & (size(p2) = (size(x) + 1) div 2)

= (let (n: integer = size(x))
  <subseq(x,1,(n div 2)), subseq(x,(n div 2) +1, n)> )

function MERGE(X1: seq(integer), X2: seq(integer)) returns seq(integer)
computed-using
true => MERGE([], Y2) = Y2,
true => MERGE(Y1,[]) = Y1,
first(Y1) <= first(Y2) =>
    MERGE(Y1,Y2) = prepend(MERGE(rest(Y1),Y2),first(Y1)),
first(Y2) <= first(Y1) =>
    MERGE(Y1,Y2) = prepend(MERGE(Y1,rest(Y2)),first(Y2)) )


function BAG-EQUAL (B1: BAG, B2: BAG) :boolean
= SEQEQUAL(SORT(B1,'<=), SORT(B2, '<=))

function SEQ-TO-BAG (X: seq(BAG-ALPHA)): BAG=X
```

The REFINE specification uses a recursive functional representation. Although this is a divide-and-conquer algorithm specification, there is no identified parallelism. However, one can easily identify where this specification can be modified to exploit parallelism. The P-REFINE version of this specification is provided later in this document.

To summarize, the general algorithm sorts a list by initially dividing the list into the smallest components and then successively merging the sublists. The parallelism is achieved through a

distribution of the sublists, with each processor performing an equivalent merge or split operation. The parallel languages PCN and UNITY show the explicit parallelism of the algorithm, whereas the parallelism is not readily apparent in the REFINE specification.

8.3 A Parallel Search Algorithm

A searching strategy is applied for a specific application to *search* the possible solution space of the problem. The solution space can be viewed as a search graph, where each arc represents part of a path to an alternative solution, and each node represents a possible solution. The relationships between the nodes sharing an arc is determined by how the search tree is generated. The searching problem can be categorized by its objective, selection, feasibility, and bounding functions.

The objective function defines the specific solution criteria for the problem. The selection function is used to select the next solution to be evaluated and directs the overall search progression. The feasibility function checks the selected solution node for feasibility of progression of the search and whether the particular node is an overall solution of the problem. The bounding function uses heuristic information to bound the number of nodes of the solution space that must be explicitly searched. The appropriate combination of these functions provides a description of the search process and determines how the search progresses. The progression of the search can take on many forms but generally can be categorized as a depth-first search, breadth-first search, branch-and-bound search, or combination of these techniques. For more detailed information on the various types and techniques applied for the searching problem, see (95).

The Best-First (BF) search was selected as the specific algorithm to model. A BF search strategy was selected because it represents a general class of search algorithms with specific activities that can be parallelized. What sets the general best-first technique apart from other search strategies is its selection of the best candidate from *all* possible candidate nodes. The overall strategy employs a heuristic function in the evaluation of the possible candidates. A *heuristic function*

is a computer algorithm that applies domain knowledge to bound the search and therefore speed up the time to an acceptable solution.

BF search techniques add heuristic knowledge to the search process. The nodes are each evaluated and ordered based upon this heuristic knowledge. A general BF search algorithm is provided below. The algorithm description provides a step-by-step calculation.

The generation of possible candidate solutions (by their selection) ensures that the search progresses as an even frontier of candidates. A general description of this algorithm is as follows (95):

1. [START] Put the start node s on a list called OPEN (of unexpanded nodes).
2. [Check Open] If OPEN is empty, exit with failure; no solution exists.
3. [Select] Remove from OPEN a node n at which f is the minimum heuristic value (break ties arbitrarily), and place it on a list called CLOSED (to be used for expanded nodes).
4. [Expand] Expand node n, generating all its successors with points back to n.
5. [Feasibility] If any of n's successors is a goal node, exit successively with the solution obtained by tracing the path along the pointers from the goal back to s.
6. [Calculate] For every successor n' of n:
   a. Calculate f(n')
   b. If n' was neither on OPEN or CLOSED, add it to OPEN. Assign the newly computed f(n') to node n'.
   c. If n' already resides on OPEN or CLOSED, compare the newly computed f(n') with the value previously assigned to n'. If the old value is lower, discard the newly generated node. If the new value is lower, substitute it for the old (n' now points back to n instead of its previous predecessor). If the matching node n' resides on CLOSED, move it back to OPEN.
7. [LOOP] Go to step 2.

NOTE: f(n) is the heuristic function applied to each node

The BF search strategies can be categorized based upon their specific search graph structure. The two major categories of BF techniques are those used with AND-OR graphs and those used with OR graphs. An AND-OR graph can be used to represent the need to accomplish two or more subtasks from a given decision point. An OR graph can be used when exactly one subtask is required from a set of mutually exclusive paths (37).

The OR graph BF algorithms can be further categorized according to two criteria: delayed termination and recursive weight functions. Delayed termination allows the searching to continue

even after a first solution is found. With a delayed termination, there is a delayed determination of the overall solution until a condition exists where no other search paths will lead to a better solution (95:55). A weight function W is recursive if for every node n in the graph, the value $W(n)$ can be calculated from the values for all immediate successors of n (95:57).

There are several possible methods to parallelize the best-first search. The general methods of parallelizing the search process are discussed first, followed by specific examples. The general parallel approaches to search strategies are dependent upon the overall objective of the search (e.g., any solution, the optimal solution, or all solutions). The strategies will also vary based upon the specific parallel programming model employed (SIMD, SPMD, MIMD). There are numerous variations that can be applied, and only a limited set are described here. The optimal solution for a specific NP-complete optimization problem is assumed to be the objective of the search. In order to find the optimal solution, we need to employ the BF algorithm with a delayed termination to ensure the solution found is optimal.

The predominant strategies usually employ a centralized controller and a set of workers. The controller is responsible for maintaining the centralized data structure for ensuring the appropriate progression of the search toward an optimal solution. The management of the data structure attempts to ensure that the explicit search is minimized. The controller also facilitates the communication to the workers, although there are variations. The workers are responsible for searching a specific area of the search space or performing the tasks assigned by the controller node.

In applying these general characteristics to the BF search strategy, two candidate parallel strategies can be derived. The first strategy uses the multiple workers to independently search portions of the search space using a BF strategy. These workers periodically synchronize and update the controller. The controller updates the overall data structures and redistributes the search assignments. This method takes advantage of the multiple processors to search the overall search space. At the processor level, a BF strategy can be easily seen; however, at the system

level, a modified BF is applied since the individual workers may expand more than just the best candidate.

A second strategy can be used to apply a system level BF strategy. In this particular application all selections and searching are performed by a centralized controller. The workers are used to assist the controller in applying the various calculations in parallel; for example, they may be used for determining the possible candidates successors, calculating its associated heuristic value (using data parallelism), and performing the alternative tasks (functional parallelism). This particular parallel search strategy is more correctly referred to as a *Parallel Best-First** because of its overall approach to searching the solution-space.

These alternatives represent both a coarse-grain (|| search) algorithm and a fine-grain (|| f(n') calculations) algorithm for BF search. The parallel search method has the advantage of quickly searching a greater portion of the solution space; however, more nodes are possibly expanded (since the best candidate is not the only node expanded at any given time). The parallel heuristic function calculation has the advantage of being a much *smarter* branch-and-bound algorithm, since it only expands the best candidate. The details of the specific problem's solution space would determine which is a better algorithm.

8.4 Example Problem Specifications using New Syntax

The following specifications demonstrate the use of these constructs for the merge sort and best first search. The specifications assume that the new parallel constructs are added to the REFINE environment, creating *P-REFINE*.

8.4.1 Parallel Merge Sort Specification.

```
function MERGESORT(Ls:seq(integer))
returns
(SLs: seq(integer) | BAG-EQUAL(SEQ-TO-BAG(Ls), SEQ-TO-BAG(SLs)) and LE-ORDERED(SLs))
% This ensures a permutation and the LE ordering
= { ?  Ls = [] -> [],
       rest(Ls) = [] -> [first(Ls)]
       size(Ls) >= 2 ->
        { || (let (var Y: tuple(seq(integer), seq(integer))))|
             Y <- SPLIT(Ls),
             Ls1 <- MERGESORT(Y.1),
             Rs1 <- MERGESORT(Y.2),
             SLs <- MERGE(Ls1,Rs1)}
  }


function SPLIT (x: seq(integer) | x~= [])
returns (p: tuple(seq(integer), seq(integer))
  | concat(p1,p2) = x & (size(p1) = size(x) div 2) & (size(p2) = (size(x) + 1) div 2)

= (let (n: integer = size(x))
  <subseq(x,1,(n div 2)), subseq(x,(n div 2) +1, n)> )

function MERGE(X1: seq(integer), X2: seq(integer)) returns seq(integer)
computed-using
true => MERGE([], Y2) = Y2,
true => MERGE(Y1,[]) = Y1,
first(Y1) <= first(Y2) =>
    MERGE(Y1,Y2) = prepend(MERGE(rest(Y1),Y2),first(Y1)),
first(Y2) <= first(Y1) =>
    MERGE(Y1,Y2) = prepend(MERGE(Y1,rest(Y2)),first(Y2)) )
% The computing-using operation could be replaced with the "?"; however, the
% pattern recognition features of this operations would need to be added.

function BAG-EQUAL (B1: BAG, B2: BAG) :boolean
= SEQEQUAL(SORT(B1,'<=), SORT(B2, '<=))
% determines whether two sequences are a permutation of each other

function SEQ-TO-BAG (X: seq(BAG-ALPHA)): BAG = X
```

8.4.2 Parallel Best-First Specification. The following specification represents a high-level specification of the Best-First strategy using the proposed parallel composition operators.

```
The Best-First Algorithm BF:
% [START]
{ ; |  Put the start node s on a list called OPEN (of unexpanded nodes),

% [Check Open]
   {; | If OPEN is empty, exit with failure; no solution exists,

% [Select]
   Remove from OPEN a node n at which f is the minimum heuristic value (break
   ties arbitrarily), and place it on a list called CLOSED (to be used for
   expanded nodes),

% [Expand]
   Expand node n, generating all its successors with points back to n,

% [Feasibility]
   {||  |(for all n') If any of n's successors is a goal node,
       exit successively with the solution obtained by tracing the path along
       the pointers from the goal back to s},

% [Calculate]
   { ; | {|| |(For every successor n' of n)
              Calculate f(n')},
         { ? |  ~(n' in OPEN) or ~(n' in CLOSED) ->
                    { || |  Add it to OPEN,
                            Assign the newly computed f(n') to node n'},
                (n' in OPEN) or (n' in CLOSED) ->
                    { ? | old_f(n') <= new_f(n') ->
                              Discard the newly generated node,
                          old_f(n') > new_f(n') ->
                             {|| | old_f(n') = new_f(n'),
                                   { ? | (old_n' in CLOSED) ->
                                         move it back to OPEN.}
                             }
                    }
         }
    },

% [LOOP]
  Go to step 2}
}
```

8.5 Chapter Summary and Conclusions

This chapter introduced general parallel sort and parallel search problems as example problems. These problems were used to clarify the possible use of the parallel constructs. It was not the intent to have these specifications be the ***best possible solution***; they are merely provided as a solution that demonstrates parallel implementation using the parallel language constructs.

These examples demonstrate how the new constructs can be used to specify an application's parallelism. The parallelism is embedded into a specific control structure (algorithm). The parallelism of the algorithm may be at a high or low level, defining either a coarse or fine grain of parallelism. The actual parallel execution is achieved by mapping the inherent parallelism to the independent processors of the system.

IX. Results, Recommendations for Future Research, and Conclusions

9.1 Introduction

We have evaluated the use of models and modeling techniques, general automated programming techniques and concepts, the issues of parallel program design (including issues of hardware architectures, algorithm design, and informal techniques for task management), and current formal specification languages and environments. This chapter describes how these concepts can be applied to a formalized environment for parallel software architecture design. Section 9.2 provides a summary of the various chapters of this document. Section 9.3 recommends the focus for future research and Section 9.4 provides research conclusions.

9.2 Summary of Results

This thesis describes the relative issues of parallel design that must be considered in the development of parallel language constructs. Specifically addressed were the relationship between the hardware architecture and software design and the integration issues faced by parallel software designers. The parallel software development process must facilitate a mapping or transformation process to the different architectures. The specification process requires a specific structuring and language syntax to support this process. Also addressed were the issues of parallel program design, specifically the problems that limit parallel program performance and techniques of solving them. The design of parallel data and control structures must address the concurrency issues, and the language used to specify these structures must include a capability to express concurrent properties.

The formal syntax for a complete set of parallel constructs provided in Chapter 7 (see Section 7.3. The *completeness* of the provided set of constructs is assumed; however, one can informally demonstrate that the constructs provide an equivalent specification capability to the constructs found in the other parallel specification languages evaluated. Other constructs may be added for convenience; the focus of this research was to provide only a *minimal set.* The general parallel sort

and parallel search problems were introduced as example problems in the previous chapter, and clarified the possible use of the parallel constructs.

Additional research is required to specify the formal operational semantics for these constructs in order to integrate them into an operational environment. We have informally described some of the issues the operational semantics would need to address. Based on a defined formal semantics, one can make assertions on the consistency of the operators. This is an important part of the overall proof process for demonstrating the usefulness of these constructs and determining their (operational semantic) correctness.

The definition of the parallel constructs applies the analysis of several existing formal specification environments and parallel algorithm design methodologies. The choice of a composition operator to represent concurrency was selected in support of all phases of design. A composition operator can be used in a graphical abstraction of the application problem's solution (such as a task system graph). The constructs can also be applied in a transformational programming environment.

The types of composition operators defined was selected to cover each area of the constructs defined by Dijkstra for control flow (sequence, iteration, and decision) (33). Through a relationship to Dijkstra's research and Church's Thesis, we have informally characterized the completeness of our set of operators.

In addition, the operational semantics for these constructs allows for the nondeterministic execution. Thus, a solution's tasks can be mapped to multiple processors with an interleaving of execution. The nondeterministic interleaving of tasks provides the parallel software designer greater flexibility in mapping to a parallel execution.

The specification of parallel applications using these constructs facilitates the mapping to multiple architectures. For example, the mathematical properties of the operators allow for reconfiguration of task graph representation. In addition, by transforming the shared variables to

channels (adding the additional constraints and control structure), the applications can be mapped to distributed parallel architectures (23).

In summary, the defined parallel constructs allow a designer to specify any parallel application. Based on the identified inherent parallelism, this formal specification of a problem solution can then be mapped to a parallel platform. The overall process can be semi-automated to assist the designer in finding an effective and efficient design. The overall process is semi-automated because the designer is required to interact with the system in performing the transformations or mappings.

9.3 Recommendations for Future Research

Investigation of the following research issues is suggested for future research:

9.3.1 Implementation of Parallel Constructs in Refine. The first stage of further research should investigate the implementation of the defined parallel composition operators through extensions to the REFINE language. The methodology of the proposed conceptual environment is dependent upon a wide-spectrum parallel transformation language. Any further research relies on the existence of such a language. From the analysis conducted for this research, the extension of the REFINE environment appears to be a suitable choice.

9.3.2 More Analysis on Formal Specification Languages and Environments. The various formal specification environments and languages were evaluated based on an informal analysis. Also, due to the scoping of the research, several languages and environments were not included. Future research should evaluate additional environments and languages, for example, graphical languages such as SISAL (111), Petri net extensions (126), and the executable CSP extension languages (57). The analysis of all these specification environments should include not only an informal analysis, but also a demonstration of a design using the most promising of these languages.

9.3.3 Language Equivalence in Concurrent Representation. Although we found that there exists an informal equivalence between the methods used by the various languages for concurrency representation, a formal proof would be valuable for this research. It would also allow the creation of a mapping between the different languages and the selected language for use with this platform. For example, the existing UNITY specification for NP-complete problems contained in the AFIT Compendium (38) could be mapped to the chosen language, allowing the reuse of this knowledge.

9.3.4 Support Tools. The automated support tools for this parallel software development environment play a fundamental role in the transformation and architecture mapping process. Future research should consider a more detailed analysis of the required tools and their required capabilities. In addition, methodologies used in the existing compiler designs share common features with the transformation programming environment. An evaluation of the language techniques applied by these compilers would provide specific techniques for creation of new transformational rules and guidelines.

9.3.5 Software Architecture Modeling. The software architecture modeling methodologies dictate the organization of the application's syntactical model and therefore are fundamental to the organization of the information within the knowledge base. Because of the important role these models play in the development process, the various software architecture models and their methods of description should be evaluated in greater depth. The evaluations should consider the models concurrent properties and their capabilities to be mapped to a parallel platform.

9.4 Research Conclusions

This research focused on the general feasibility of the proposed parallel software development environment, the requirements for this environment, and an evaluation of an appropriate formal specification language for representing parallel applications. The information presented meets these

goals. With the added parallel language constructs, the conceptual parallel software development environment described in Chapter 3 should be able to represent parallel applications.

The realization of this design would require a combination of many different development methodologies and tools; specifically, the parallel design concepts embodied in the PCN, REFINE, CIP-L, and PARSA systems could be combined. The REFINE environment provided the greatest breadth of formal constructs, with high-level data types and operations ranging from functions and sets to first-order predicate calculus operators. As described in Chapter 7, the formal language for the proposed parallel software development environment can be realized through an extension to the REFINE environment. This would require the addition of parallel composition, nondeterministic choice, and a special class of protected shared variables. The creation of these extensions can apply the design philosophies used in the construct's descriptions in PCN and CIP-L. Finally, the definition of the architecture mapping support tools could apply the concepts used in the PARSA environment.

The scheduling and mapping techniques discussed in Appendices D and E would be encoded into both domain-specific and domain-independent rules and transformations available in the knowledge base. Also, specific tools for generating a model of the parallel architecture should be available. In addition, performance feedback and simulation tools would assist in the fine tuning of the transformation and mapping process.

The effort to create the described environment is no small task. There are several difficulties that would need to be overcome. These include the structure of the knowledge within the knowledge base and the correct implementation of constructs to maintain their operational semantics.

The structuring of the knowledge base abstractions would require the evaluation of several considerations. The selection of the domain-specific language and software architecture description language is critical for defining the interaction of programming and problem knowledge, and would impact the organization and representation of that knowledge. In addition, the user interface for

the platform would impact the knowledge's organization. Future research would also need to focus on modeling alternative application domains and applying alternative software architecture models to formally demonstrate the applicability of the envisioned parallel software development platform.

The implementation of the parallel constructs would require a significant amount of further research. Although the operational semantics for the constructs have been initially defined, this must be further integrated with the REFINE operational semantics to validate correctness of their execution behavior. The design effort would also require the creation of a specific execution controller or monitor to regulate the concurrent execution of statements. In addition. the operator precedence would need to be modified to incorporate the precedence of the new operators. The overall conceptual design of the extensions would then need to be validated.

The computational challenges of the 21st century require greater computer performance. Current parallel systems have taken great strides at obtaining this increased performance. Unfortunately, the complexity of parallel software design and the parallel architecture mapping problem often make it difficult to efficiently and effectively use these systems to their full potential. Through a sophisticated parallel software design process, with the assistance of automated tools, the parallel design burdens can be reduced. This thesis has provided a starting point for the creation of a transformational model and automated parallel software development environment, applying domain-specific knowledge, parallel software architecture concepts, and knowledge-based software engineering.

Appendix A. Software Modeling, Specification, and Design

A.1 Introduction

Underlying the development process and design methods are concepts of software design, algorithm design, and software architecture. To be able to integrate the various techniques for a common process, one needs to understand the relationships between the various models as they apply to the underlying software and hardware structures. An understanding of the descriptive and mathematical properties of the various models used for parallel software design is also helpful.

This appendix is divided into four sections. This section provides an introduction to the chapter and an overview of topics covered. Section A.2 provides an overview of the evolution of modeling methods, their relation to specification approaches and theories, and how problem identification impacts modeling methods. Section A.3 describes the evolution of modeling techniques within software engineering. Various methods of modeling software systems are discussed. Finally, Section A.4 summarizes the appendix.

A.2 Models

A.2.1 Mathematical Models. A model is used to precisely define the desired characteristics and properties of a system (hardware or software). The model abstracts the details of the problem, capturing the system's characteristics as specific properties of the model. The model provides the foundation for verifying the desired properties and characteristics. Designers use different models to specify different properties, each model with its own inherent power of expression. The designer must select a model that captures the desired system behavior.

In his PhD dissertation Greenspan provides several useful definitions as applied to models. A *property* is a directed relationship between one object and another, in terms of which the first is defined. Aggregation allows one to view an object as a collection, or *aggregate*, of the objects to which it is related by properties. Classification allows one to group objects that share common

properties into a *class*. A specific member of a class is called an *instance*. *Generalization*, and its converse *specialization*, allow for organizing the classes into a hierarchy from the most general to the most specialized. One important property resulting from this hierarchical ordering is the specialized relationship of *inheritance*, whereby object classes inherit attributes of their super-class. These principles became fundamental in Greenspan's development of the formal requirements modeling language RML (55) and are equally important to our understanding of mathematical models.

The various models used for engineering design are based on mathematical theories and principles. The mathematical theories and principles are used to characterize the various properties of the model. The models are created using a specific modeling language and grammar. Properties of the model become constructs of the modeling language, formally defined by the mathematical theories and concepts. A model with this strong relationship to formal mathematical principles can be referred to as a *mathematical model*.

The constructs of the language used to specify the model determines its expressive power. The expressive power of the model is also embedded in the inherent structure of the model. A model description may consist of several related, encapsulated forms. The relationship between these encapsulations provides a specific problem view and structure and may be considered a conceptual architecture. For example, a model may consist of several related sub-models and may be incorporated into a hierarchical relationship where a specialized aggregate or inheritance relationship may apply.

Many researchers of computational theory have studied the hierarchy of languages and grammars (84, 60). Since models are based on a specific modeling language, language concepts and theories encompassed in computational theory can also be applied to models. The mathematical properties, concepts, and relationships between models becomes important when trying to characterize and use the various models. The formality of a model is based on its inherent relationship to formal mathematical structures.

A.2.2 Problem Identification and Specification. A specification can also be viewed as a syntactical model. Depending on the specification language used, it may be characterized as formal or informal (a distinction that will be more completely defined in Section 2.3). The constructs available within the specification language define its expressive power and formality. For formal specifications, we may associate various mathematical theories and concepts to them. For example, a designer may formally prove the existence of an equivalence relation between alternative specifications, which becomes important in our transformation and refinement process. Additionally, the equivalence of various language constructs or groups of constructs may be determined. Specific properties for concurrency may also be included or added to the specification models.

In specifying the problem, the designer assembles the language constructs into an architectural model. The alternatives for problem identification and description are based on a given *problem view*. The similarities and differences become entangled in the evaluation of the chosen model for representation. The various models can have a range of underlying mathematical properties, theories, and structures. Examples of the varying mathematical concepts that may be included in the modeling language are:

As described in (94)

- sets, relations, mappings, functions,
- ordering and lattice structures,
- algebraic structures (e.g. groups, rings, fields, sequences, bags, trees),
- relational structures (e.g. graphs, Petri nets),
- formal systems (e.g. equational systems, grammars, automata, deduction and inference systems, state machines),
- differential equations,
- stochastic models,
- topological and geometric structures

The models chosen for description and their underlying mathematical properties impact the eventual encodings of the problem into the software architecture. For example, properties such as the minima or maxima in orderings, or the paths, cycles, or closures in relational structures

can be formally defined and used in logical analysis of the problem. Each property has a specific mathematical definition within its model structure, and a mapping to an interpretation of that property for the problem domain.

The problems and strategies can be further classified with respect to specific programming views: function oriented, data structure oriented, event oriented, control flow oriented, or data flow oriented. How the problem is perceived and categorized into these views will dictate problem decomposition.

Much of the literature in this area is divided according to classes and concepts of problems. Other works are organized based on the method or technique used for finding solutions. Both categorizations play a major role in defining software and algorithm design and thus the software architecture. The designer, based on the relationship for identification of the problem and existing models, may reuse existing design approaches and methods. By recognizing patterns and similarities between models, a designer can gain a better understanding of the problem domain and reapply prior research. The knowledge base representations are also impacted by the design decisions made in identifying the problem. Each problem type and method can be represented using different data and control structures, with different encodings in the computer. The designer must weigh the time/space trade-offs for each structure.

For example, the graph structure is used as an abstraction for representation of several problem areas. A typical graph is shown in Figure A.1. There are alternative methods for the representation of the structure of the graph. Figure A.2 a) uses a pair of sets, b) uses a set and a pair of functions, and c) uses a matrix representation. These show just a few of the alternatives. Each has an associated space/time attribute for accessing the knowledge encoded in the data structure. How the problem maps to the graph structure can also offer many alternatives.

Using the example as a city map, one could associate the nodes with street intersections, and edges with streets. Alternatively, one might associate streets with nodes that are connected by an

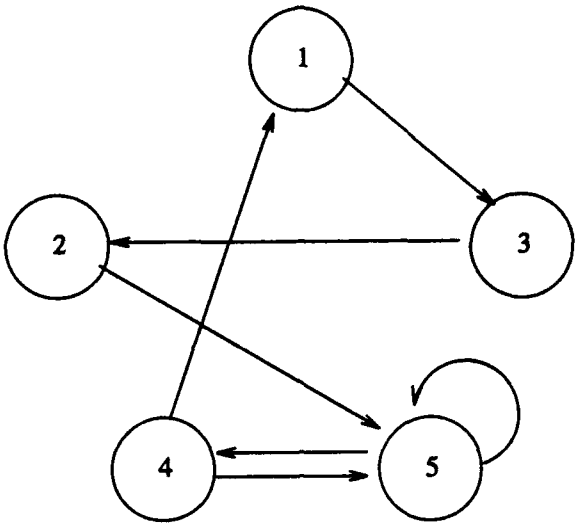

Figure A.1 Example Graph

a) a set of nodes and set of edges:

({1,2,3,4,5}, {(1,3),(2,5),(3,2), (4,1), (4,5), (5,4), (5,5)}

b) a set of nodes and pair of incidence functions i and o:

({1,2,3,4,5}) i: 1 ⟼ {4} o: 1 ⟼ {3}
2 ⟼ {3} 2 ⟼ {5}
3 ⟼ {5} 3 ⟼ {2}
4 ⟼ {5} 4 ⟼ {1,5}
5 ⟼ {2,4,5} 5 ⟼ {4,5}

c) an adjacency matrix, where component (i,j) has value 1 for each edge from i to j:

	1	2	3	4	5
1	0	0	1	0	0
2	0	0	0	0	1
3	0	1	0	0	0
4	1	0	0	0	1
5	0	0	0	1	1

Figure A.2 Alternative Graph Encodings

edge if they intersect. Which representation is chosen would depend on the nature of the problem and the knowledge required to be accessed. For the first example, it is easy to check how many streets are involved in an intersection, but more difficult to trace an entire street. A street becomes a path of the graph. In the second alternative, a given street's access to other streets is shown, but a pathway through the streets is not available. The impact of alternative choice in knowledge representation can clearly be seen by these alternatives.

As another example, a designer may encode a binary tree as a linked list or an array. The underlying computer hardware may facilitate easier access for linked list or arrays. Each of these alternatives represent the trade-offs for the optimization transformations when a designer considers implementation. The structures chosen for design can be impacted by the specification decisions made for modeling the problem. In summary, mathematical models are used throughout engineering design (55, 77, 36, 78). A given model has its own inherent mathematical properties dictated by the underlying modeling language used to encode and describe the model. The available constructs and syntax of the modeling language dictate its power of expression. A specification may be viewed as a syntactical model. When a designer encodes a problem into a given model, he is constrained by the power of the model for representing the properties and behavior of the problem. The choices he make among alternative methods of identification of the objects and behaviors for a given problem also dictate the effectiveness of the model for specification. This problem of knowledge representation is inherent in all engineering design. If a specific property or behavior of the application is of interest to the designer, he must ensure his model (or specification) captures the property or behavior. If we are to use models for the specification of concurrency, then properties of concurrency must be included within the model.

A.3 Models in Software Engineering

This section defines several different modeling techniques and properties of the application these models abstract. The various models present alternative problem views, allowing the designer to analyze specific properties and characteristics of the application. Specifically, models for describing and representing the problem with functional model, functional hierarchy, state-based description, and graphical techniques are presented.

The designer uses the models to capture the static and dynamic behaviors of the application. The designer attempts to encode these behaviors into a computational model that represents a specific software architecture depiction of the problem. The specific architecture model defines the data and control structures for the problem process.

A.3.1 Functional Model. In mathematics, to define a function, one must specify three things: an input domain, an output domain, and rules for transforming or mapping the inputs to the outputs. The mapping must always produce the same outputs for the same input data set. Applying a functional concept to design, the designer must characterize the system based on a set of input parameters, a set of output parameters, and the relationship of the input set to the output set. The designer functionally decomposes the specification into lower levels of functionality, until the lowest level arithmetic operators are encountered. The designer must use some method to determine and maintain the desired consistency and correctness between the functional levels of the hierarchy. The process decomposes both the control and data flow simultaneously.

The specification describes the functional relationships, whereas the design specifies an algorithm for determining specifically *how* to compute the inputs to obtain the desired outputs. Multiple algorithms could achieve the same desired functional relationship but obtain varying degrees of accuracy. For example, there are multiple methods of computing the sinusoidal value for a given x. Each implementation has its own inherent accuracy and time/space relationships. But to be considered correct, each alternative must meet the functional level specification.

A functional description can also be represented using a bipartite graph. Each node represents an element of the domain or range, with arcs representing the functional mapping between the nodes of the domain and range. Functions having equivalent graphs would be considered equivalent (122).

Taken by itself, the functional approach is limited due to the lack of inherent memory in mathematical functions. Mathematicians have attempted to overcome this limitation through the use of recursive methods; however, recursion increases the complexity of the functional description.

Another limitation of a pure mathematical functional viewpoint is the considerations for partial functions. Most computer languages have made a trade-off to increase their expressive power. The constructs of the language allow for the possibility of infinite looping, while adding a more powerful control structure. The languages also allow for the inclusion of partial function definitions. When performing formal proofs, the inclusion of partial functions and the halting problem creates an unsolvable problem unless restrictions are used. Even with restrictions, the proof remains difficult at best.

Because of these limitations, functional description for representation of properties of software design is a necessary but insufficient property for concurrent specification(5).

A.3.2 State Machine Model. The designer can extend the model to include a concept of *state*. Thus the software architecture models a finite state machine representation where, not only are the set of inputs and outputs modeled, but also a set of states and functions for state transition and output. The new functional representation determines the output based on both the input and current state. A function also may characterize the state change via a transition function. The transition function determines *next state* based on the input and current state and may determine an output.

There are two fundamental problems with the state machine model extension as applied to concurrent system specification. First, the model inherently serializes the underlying concurrency by assuming a single, finite sequence of actions. The model assumes the processing on an input is

completed prior to the next input. This may result in states in which two or more arrivals occur at the same time. The correctness proof for the model may overlook this possibility. Communicating sequential processes (CSP) overcomes this problem by assuming multiple finite state machines operating in parallel. The second fundamental limitation is that the state machine model is flat with no inherent way of handling complexity. The designer must add his own additional structuring for understandability, resulting a complexity problem (5).

A.3.3 Functional Hierarchy. The structuring which offers the greatest flexibility, as well as understanding, is the functional hierarchy. Within a functional hierarchy, the designer successively decomposes a functional description into lower-level primitive functions creating a hierarchy of functions. The breakdown between levels of functions generates a level structure with functional relations for input and output. The various levels of view contain an inherent software architecture. The concept of functional hierarchy is fundamental to all types of engineering design. For example, relationships of aggregation and inheritance are expressed through hierarchies.

The functional programming model described thus far provides a structured view for specification. However, in regard to the concurrent (especially the distributed) model, it provides a necessary but not sufficient representation. The conceptual view of the sequence over time of a series of inputs is not distinguishable. For example: a given function F may receive as input; X and Y, X or Y, or X followed by Y. The model cannot differentiate representation between these sequence variations. This lack of temporal properties limits its ability to specify concurrency. Also, when the analyst decomposes a function G into sub-functions G_1, G_2 and G_3, the model cannot capture the ordered sequencing or concurrency of the subfunctions execution (5). A specific control structure is required for this but is not represented in the functional model. The model is therefore insufficient for the representation of concurrent control flow. A solution, as we will see described in the next section, is to combine and extend a graphical model of a network flow with the functional and finite state models.

A.3.4 Graphical Modeling. The graphical model uses a directed graph to specify either the control or data flow of the system, or both. There are several different graphical models used throughout computer software design. Graphical modeling techniques are used in Petri nets, object-oriented design, and finite state machine descriptions, as well as other modeling techniques. The advantage of the graphical model is the intuitive understanding for the designer. The graphical model provides a powerful abstract and allows for visualization of the problem. Graph theory has been extensively studied and can be applied to these modeling techniques (28).

By combining the control flow concepts of Petri nets with aspects of both object-oriented and functional modeling, a complete system perspective can be presented. Each node can represent a state change within the system, with each arc providing state transition and data dependencies. The designer can structure the nodes based on the problem view he uses in decomposition and specification of the problem.

Underlying the graphical model is the graphical language used to encode the graph. The formalism of the graphical language used in the encoding has a direct relationship to the formalness of the model. In his PhD dissertation, Bailor describes various methods of formalizing graphical modeling languages for parallel software design (7). The formalism allows for logical reasoning based upon the graphical and mathematical theories. Other researchers have also shown the advantages of graphical modeling methods (28, 26, 65, 80, 97).

A task graph is a primary example of the graphical approach to modeling which includes aspects of: functions, functional hierarchies, state-based design, control flow (Petri nets), and concurrency.

A.4 Summary

The development processes of today rely heavily on the diverse power of description, encapsulation, abstraction, and mappings provided through modeling. Combining mathematical concepts

with the structured design concepts provides improvements to the design process through their logical hierarchy and formalization. Examples in logic, functional, and object-oriented programming demonstrate the capabilities of the alternative structures within models. Domain-specific concepts and software architectural concepts provide an organization and structure that facilitates mapping between the problem and the computational machine. The static and dynamic behaviors of the system, as well as the control and data flow, are captured in the various models used in development today. In addition, properties of concurrency can be explicitly or implicitly captured within a given model. The model is encoded into the computer through a mapping between the modeling language and a computer language.

Appendix B. Parallel Performance Issues

B.1 Introduction

There are several terms used to describe parallel performance and concurrent properties. This appendix defines the various performance measurements in Section B.2. Section B.3 describes several factors that can limit parallel performance. Finally, Section B.4 defines several of the terms used to describe formal properties of software and algorithm design in the parallel environment. The subject is discussed in greater detail in (38).

B.2 Parallel Performance Measurements

B.2.1 Background. Early computers were designed with an inherent "von Neumann Bottleneck." Computers in the 1950s had a single control unit and one bus connecting memory to the control unit. This single processing element (PE) formed a choke-point through which all actions taken by the computer had to flow. Builders of the big mainframe computers partly overcame this by adding smaller processors to handle disk access and input-output. Parallel computers take the solution even further by adding many more coprocessing elements.

Minsky's Conjecture, in the late 1960s, claimed the performance of parallel computers would be limited by $\log p$ due to the communication overhead between processors. His research was based on experience with a limited number of processors and the conventional operating systems of this time. This rather pessimistic view was abandoned as a result of better operating system design and engineering. Multitasking algorithms were developed, and communication and hardware architectures have improved tremendously.

The potential speedup using a parallel, von-Neumann based computer can be defined as

$$S_p = \frac{T_1}{T_p} \tag{B.1}$$

where T_1 = execution time for the best serial algorithm on a single processor and T_p = execution time for a parallel algorithm using p processors. Thus, if a sequential algorithm to simulate a VLSI circuit through 10 seconds of activity takes 120 minutes to complete when implemented on a single processor, applying the same simulation using an effective parallel algorithm on a parallel computer with 120 processors would theoretically take 1 minute! Unfortunately, it was suggested by Amdahl that program speedup was limited.

Ahmdahl's Conjecture, also in the late 1960s, stated that the overhead to the parallelism is the inherently serial portion of the process. This overhead, it was believed, places an upper limit on throughput of five to seven times sequential processing rate. Stated succinctly, Ahmdahl's claim was that, if s is the fraction of the program that must be done serially, even an infinite number of processors could only produce speedup of $\frac{1}{s}$

Let s be the fraction of the program that is naturally serial. The remainder $(1-s)$ of the program is naturally parallel. If $T*$ is the best parallel algorithm, then the serial part takes $s \times T*(n)$ to compute; the parallel part takes $\frac{(1-s)\times T*(n)}{p}$ time. The total time to compute the solution with a parallel machine would then be the sum of these two. The speedup, as given above, then amounts to: $\frac{p}{(s\times p+(1-s))}$

This view is very pessimistic: $\lim_{p\to\infty} speedup = \frac{1}{s}$. It ignores the possibility of a new algorithm with much smaller s. Such a new algorithm might not be more efficient on a single processor; that is, we may already have had the best serial algorithm. The new algorithm need only have a smaller fraction of its work being inherently serial. Amdahl also ignores the possibility of more of the program being run from higher speed memory.

A more optimistic, and realistic, view of parallel programs is known as the Gustafson-Barsis Conjecture from the late 1980s. They pointed out that often the problem is scaled with the number of processors, and the serial part is a constant amount of work. Thus a jump from 10 to 100 processors simultaneous with a move from 12 to 102 units of work (2 units being the serial part),

provides much better than $\log p$ speedup. Using this conjecture, $\lim_{problem\ size \to \infty} s = 0$. But p is scaled beyond the problem size; the result is a drop-off of average efficiency. Other problems may fall into this category by meeting this criteria if the growth of the serial portion of the execution is much less than that which can be parallelized.

Amdahl did not consider that as the problem size scales up, the inherently serial part drops as a percentage of the overall program. Of course, it might be that the inherently serial portion also scales up with problem size. For problems where the serial portion grows faster than parallel portion, Amdahl's conjecture applies. The inherent characteristics of a problem must be examined before an attempt is made to implement it on a parallel computer. The designer must identify up front the inherent level of parallelism within the problem to generate an effective parallel algorithm (38).

Within this section, several of the key performance characteristics for parallel design are described. These performance parameters encompass the measurements used for parallel algorithm, software, and hardware design. The traits discussed include: execution time, grain-size, scalability, speedup, efficiency, and utilization.

B.2.2 Execution Time. The execution time for a given process measures wall clock time to complete the execution of the program. It includes the time to execute the program instructions, the memory access, communication times, and other overhead processing. The primary objective in any programming exercise is to reduce execution time, thus gaining speedup. A given design includes parallel tasks, which can be executed concurrently. However, these tasks may include interdependencies requiring synchronization or the exchange of data. The tasks may also share certain hardware resources causing contention and delays (system bottlenecks). The execution time includes all these overhead parameters in its calculation. A parallel design seeks to exploit the number of tasks that can execute concurrently while minimizing the effect on execution time for the overhead of synchronization and data transfer.

B.2.3 Grain-Size. Grain-size is a parameter that has been used to characterize both architectures and applications. The challenge to the designer is matching the grain-size of the application to the architecture (or vice versa). The underlying feature of grain-size is the ratio of the communication costs to the computational costs. A designer decomposes an application into a group of interacting tasks, which may include data and synchronization dependencies. A designer can then informally characterize the application for grain-size based on the ratio of the task size to required communications. However, the actual grain-size cannot accurately be determined unless there exists a mapping between the application and a given architecture, where actual values for the communication latency costs exist. The calculation of grain-size is relatively meaningless without this mapping and is thus informal.

A coarse-grain application is one that decomposes into pieces that can work independently without interaction for long periods of time. A fine-grain application is one whose decomposed pieces require more interaction. The extreme of coarse-grain problems are those that are embarrassingly parallel: the pieces can be computed entirely separately. An extremely fine-grain problem is one that should be kept serial: the communications overhead of the massive interactions is too high to permit gain from parallelization. However, a given application can have any number of decompositions based on alternative design decisions. The designer should decompose from a top-down perspective and strive to initially decompose the application into the finest grain possible. The tasks can then be grouped to reduce interprocess communication. This provides more options in future mappings.

In classifying architectures there are varying guidelines provided on grain-size. An architecture with relatively fast, small processors is typically considered fine-grain, whereas a system with large, complex processors is considered coarse-grain. However, the advances in VLSI design and Reduced Instruction Set Computer (RISC) design has widened the spectrum of alternatives, making determination of grain-size relative. What determines small vs. large? All architecture

designs attempt to keep communication costs at a minimum, but the ratio of the communication to computation is the underlying feature for determining grain-size.

Thus, the designer tries to match the application, its decomposition and partitioning, to a given architecture. The objective is to balance the communication and computational costs to increase efficiency and reduce latency. The impact of the decomposition, partitioning, and mapping can have dramatic effects on performance. The appropriate balance between the communications and computations can not only impact the execution time but the scalability as well.

B.2.4 Scalability. The question of scalability is of great concern in parallel algorithms but is often treated with only a surface-level consideration in discussions of parallel programming. The driving force behind using the massively parallel processor (MPP) environment over other supercomputing models is the issue of scalability. Problems on MPP systems tend to scale toward the realization of greater speedup than on other systems (38).

Fortunately, parallel applications tend to scale with the available computing power, *i.e.*, given more computing power with a parallel computer, the applications are expanded (more data items, for example) to utilize the available hardware resources. For instance, doubling the number of processors allows doubling the number of data variables to ensure comparable utilization of the physical system. The serial portion of the code tends to scale linearly with respect to the concurrent code, allowing for scalability of application (38).

Scalability is an issue of performance that is affected by the decomposition, partitioning, allocation, and scheduling decisions made for a particular architectural implementation. The scalability of a problem for a given architecture is highly dependent on an effective mapping strategy that balances the communication to computation ratio for the entire task system. The adequate matching of grain sizes between the application and the architecture is primarily influenced by the decomposition and clustering of tasks. Therefore, any design must begin by decomposing the problem into the finest grain possible for a complete representation of the problem. The designer

then can apply various clustering techniques to tailor for a specific implementation. Each of the possible clusterings represents an alternative mapping problem and an alternative schedule.

Singh and Hennessy (115) point out that few, if any, programs can be scaled up for an indefinite number of processors. This point has been largely ignored in the literature. We tend to think of scalability only in terms of input data size, when we should also concern ourselves with numerical accuracy, interval between time-steps, and other variables. They call the emphasis on input data size *naive* scaling, and the more comprehensive view *realistic* scaling. More comprehensive methods must be developed to look at *how* an application should be scaled (38).

B.2.5 Speedup. Speedup is defined as the execution time of *the best* serial algorithm divided by the time to execute the parallel algorithm on p processors, for a problem of size n. Speedup depends on the ease of parallelization of the algorithm and the number and computation power of the processors. If the algorithm is not parallelizable, no speedup can be achieved. Likewise, as p increases, it is possible that no further performance gain (i.e. a decrease in execution time) is possible due to domination of the problem's overhead.

Designers use speedup graphs to display the effective speedup for a given algorithm design. A 45 deg line, $\frac{1}{p}$, is defined as *linear speedup.* Linear speedup is the objective for parallelizing an algorithm. Realized speedup can be below the linear value due to suboptimal algorithms or domination of the overhead processing, that is, if the best serial algorithm is directly implemented when the best serial algorithm is not the best parallel algorithm. Speedup can also be reduced to algorithm overhead (from an inherently serial portion of the algorithm), improper load balancing, and communications overhead. Speedup also depends on granularity of decomposition and problem size as associated with the communications overhead. That is, if the implementation is too fine-grained, the communications becomes the dominating factor in the overall execution time (38).

Superlinear speedup is defined as obtaining *speedup* $> p$ or *efficiency* > 1. When the curve shows the parallel algorithm running faster on one processing element than the *best* serial algorithm,

the implication is that the serial algorithm is not truly the best. Often, an attempt to parallelize the algorithm helps us find a better serial algorithm.Other times, the parallel computer solves a slightly different, easier problem, or is provides a slightly different answer.

But superlinear speedup above a single processing element is possible, because a parallel computer usually has p times as much cache and main memory, and p times as many registers as a serial computer. Thus as the separate processing elements process the data, more operations obtain cache hits, more data can be in main memory (rather than having to be fetched from disk), and more operations can involve registers (rather than main memory). Likewise, the communications network can provide some computation in the form of combining logic. In such cases, there are effectively more than p processors (38).

B.2.6 Efficiency. A measure of the efficient use of the processors used in computation is referred to as *efficiency*. The designer calculates efficiency by dividing speedup by the number of processors, p. For example, we might get a speedup of 4, but if it takes 8 processors to do so, the efficiency is only 0.5. Thus, efficiency measures the useful work of the processors at solving a problem. Issues of overhead processing impact the relative efficiency parameter, and therefore factors of overhead processing should be considered in determining effective efficiency (38).

B.2.7 Utilization. Processor utilization measures the utility of each processor for solving the problem. The designer calculates processor utilization by dividing the compute time by compute time plus idle time. Each processor will likely have a different utilization. The designer should be aware of the overhead processing when calculating the effective utilization. Certain management control features may increase processor utilization while not contributing to reducing the overall execution time for obtaining a solution. Because overhead is included in execution time calculation, the execution time may increase even though a designer has increased processor utilization – thus the effective efficiency has been decreased!

B.3 Factors that Limit Parallel Performance

When designing a parallel algorithm, constructing a software architectural model, and mapping this to a specific parallel machine, the designer needs to consider the various factors that may limit parallel performance. Parallel program performance may be limited by: memory contention, excessive sequential code, centralized control, communication delay, synchronization delay, load imbalance, and overhead processing.

Memory Contention and Delays

The problem of memory contention exists for applications on shared memory computers. Processes use shared variables to communicate and synchronize, and as global variables. The system memory becomes the system bottleneck. Several processors try to access memory simultaneously, requiring some processors to wait. In addition, delays for memory access can exist because of an inadequate cache structure.

Excessive Sequential Code

Amdahl discovered the limitation on speedup created by the inherently sequential portions of some algorithms. In some of these algorithms the growth in complexity of the sequential code can dominate the overall algorithm.

Centralized Control

Sequential code can also be introduced by the use of a centralized controller. The centralized operations may become a system bottleneck and limit the maximum overall speedup that can be achieved.

Communication Delays

Communication delays may be encountered for many different reasons. In the distributed multicomputer architecture, the processors interact through message passing. The communication interconnection network must support the required communication structure of an application to be effective. Contention for communication channels can cause delays in the passing of information.

The locality of the allocation of processes to processors has a dramatic impact on the experienced communication delay.

Synchronization Delay

In order for the processes of the system to synchronize, one or more processes may need to *block* or wait. This is also the case when the processes rely on each other due to a data dependence. The effect may be to limit the overall performance.

Load Imbalance

A load imbalance results from an unequal distribution of the overall application workload. The specifics of the load balance problem and possible solutions are identified in Appendix D.

Overhead Processing

There are various *overhead* tasks that can be included as part of the processing of a given application. These may be due to the operating system processing or the control structures generated to overcome some of the other problems identified above. One example is the time to create processes in a dynamic environment. Another example is the processing required for the control required for an algorithm to overcome load imbalance.

B.4 Formal Properties of Design

The reasoning and analysis of specifications relies on the use of a formal logic system. For concurrent applications this requires a formal system incorporating the domain of time. By expanding predicate calculus to the formal system of temporal logic, one can reason about specific properties of the application specification. In addition, properties and characteristics of concurrent design can be defined and used in the reasoning process. This section informally defines several of the characteristics and properties of concurrent software design.

A task is an encapsulation of an execution entity. Tasks may be defined in terms of functionality, visible events only, or as states with state transitions. Also to be specified are means for

starting, suspending, resuming, and terminating parallel activities. Progress toward a terminating condition must be specified. A method must exist for the communication among tasks, whether through messages or shared variables.

In identifying and defining the tasks of an application, certain characteristics of interest should be specified:

actions What should the tasks do? What is their dynamic behavior?

choices How do the tasks choose what actions to take? What is the process of selection between alternatives?

fairness Do all tasks have an opportunity to execute?

sequencing What is the ordering in which to perform the actions? How is this dependence represented within the control structure?

scheduling What is the best order in which to execute tasks?

concurrency What is the maximum *beneficial* number of actions that can take place in parallel? The dependence and independence of the tasks define the relationships of interest for defining this feature.

synchronization How often must tasks rendezvous, and how efficiently can this take place? Should the rendezvous be synchronous, loosely synchronous or asynchronous? The required synchronization represents a dependency relation between tasks.

likelihood How can the stochastic characteristics of the system be best modeled?

exceptions How should unusual circumstances be handled? Should the constraints of specification preclude exceptions?

time/physical constraints What is the computational complexity of the algorithm and how does it relate to the resource consumption and execution time?

Use of a formal system often permits properties of interest to be automatically analyzed. The logical properties of the design include:

correctness Accurate description of desired behavior.

liveness Specification free of deadlock conditions.

boundedness Resource constraints are met; State transitions finite.

persistence Specific events are never permanently disabled.

fairness No task is ever permanently prevented from executing. Equal likelihood among choices.

synchrony Conflicting events do not occur simultaneously.

Performance properties of a design require support for stochastic analysis. The properties listed are characterized based upon a system design — an execution model consisting of both a software and hardware architecture with a mapping between.

throughput Number of transactions processed.
cycle time Time between transactions.
event to event time Time between two distinct events.
recurrence rate Frequency at which an event recurs.
resource consumption Number of resource units needed over time.
utilization ratio Portion of time a resource or state is occupied.

For each of these, a mean, maximum, and minimum value is desired.

Appendix C. Parallel Architectures

C.1 Introduction

Computer architecture has evolved significantly over the last decade with the proliferation of parallel computer research and development efforts. The driving force behind such investments is the potential speedup of solving complex and computationally intensive problems such as wave mechanics, fluid dynamics, structural analysis, Very Large Scale Integration (VLSI) circuit simulations, weather prediction, chemical reaction simulations, image processing in computer graphics, and real-time assignment based problems within the Strategic Defense Initiative (SDI) research area (79, 113). VLSI technology for the design of a single processing element has neared its peak. Only through a parallel computing paradigm can additional programming speedup be achieved.

The need for faster computers is quite apparent. Weather prediction results are useless if the time to produce the prediction is greater than the time until the actual event occurs, *i.e.*, the real-time constraint cannot be met. As another example of the need for further improvements, the solution of the Naiver-Stokes equations used to aid in aircraft design is currently performed on the largest available computers, yet it still obtains only an approximate solution. The same is true of SDI research involving real-time assignment strategies of weapons to targets. A solution to this problem is equally useless if the time it takes to produce the solution is longer than the time for the incoming missiles to reach their point of destruction. These are just some examples of application areas that can benefit from increased parallel performance.

The basic concept behind parallel computing is to have more than one processor involved in the computation of the problem. Parallel computing can use anywhere from two to several thousand processors operating on a given problem. As discussed in Appendix B, the designer's objective is to obtain speed-up relative to the number of additional processors. However, this is not always achievable because of the overhead for communication and synchronization of tasks. In addition, the application may be limited by the percentages of inherently sequential versus parallel

operations. In designing software for a parallel environment, the algorithm design may require alternative approaches to those used in sequential design.

This appendix describes the alternative parallel processing models (SIMD, SPMD, and MIMD) and the alternative architecture models (multicomputer and multiprocessor). The implications of these alternatives are summarized in Section C.4.

C.2 Parallel Processing Models

There are several parallel programming/architectural paradigms to choose from. The common classifications are: single instruction, multiple data (SIMD); single program, multiple data (SPMD); or multiple instruction, multiple data (MIMD). Typically, designers use these characterizations to describe the hardware. Because of the flexibility of current architectures, the system may support several of these options. The decision to use SIMD, MIMD, or SPMD will determine the approaches taken in the design of the software and choices in portability among architectures.

The variations in these paradigms are primarily due to the underlying control structure embedded within them. While SIMD relies on centralized synchronization control, MIMD distributes the control throughout the system. SPMD, on the other hand, synchronizes based on a series of instructions (a program). Researchers have performed various studies to determine which paradigm is best for the various applications (47, 101, 32).

The applicability of the various processing models to a specific architecture is dependent upon the control structure(s) supported. The particular processing model selected may be impacted by the interconnection topology of the machine due to its impact on inter-process communication and memory access. The designer must select an appropriate method for embedding the selected processing model into the software design. The selection of the appropriate processing model is therefore not only dependent upon the particular application, but also dependent upon the characteristics of the implementation architecture.

C.3 Multiprocessor versus Multicomputer

This section describes the two predominant parallel architecture models. We describe each of the architecture models and the performance issues these architectures must overcome to operate effectively.

There currently exist several variations of parallel architecture: data flow machines, reduction machines, and several variations of Von Neumann based machines. A categorization and hierarchy of architectures is found in (32). Each architecture may be characterized by its interconnection topology and its grain-size. The Von Neumann architectures may be further characterized by being either a multiprocessor or multicomputer. The interconnection topology defines the computer's communication and control structure. The execution and latency cost of the *computation to communication ratio* determines the grain-size. Because there are currently no commercially viable data flow or reduction machines, these were not included in the categorizations (although one may view these as sophisticated fine-grain, distributed computers). We clarify the distinctions between multiprocessor and multicomputer further below.

C.3.1 Multiprocessor. The multiprocessor is characterized by the individual processors having access to a common shared memory. The shared memory allows for shared use of various data values and data structures. The processors use the shared memory for interprocessor communication and synchronization. Individual architectures have variations in their connection topology, although the common bus structure is the most popular. The common bus limits the number of processors based on its saturation point for bus bandwidth, thus limiting the architecture's scalability. Because of the common memory for all processors, there are also limitations due to memory contention. Several processors may attempt to access memory simultaneously, and some processors will be required to wait, while other requests are satisfied. Figures C.1 and C.2 show example architectures.

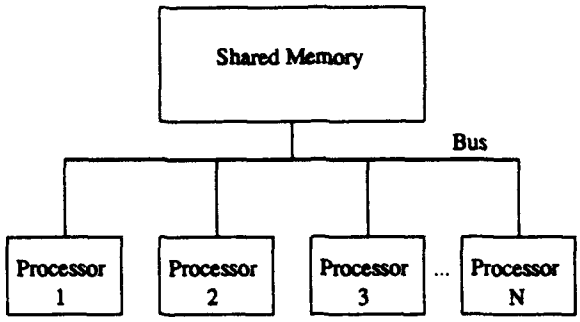

Figure C.1 Multiprocessor Architecture with Common Bus

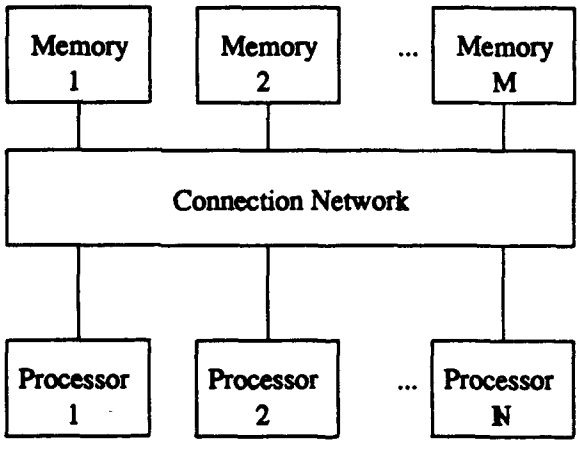

Figure C.2 Shared Memory with Multiple Memory Modules

Architecture designers have implemented several techniques for overcoming these limitations. Variations include memory caching, memory partitioning, and alternative bus structure connections.

1. *Cache Memory.* Caching provides each individual processor with its own local cache memory, decreasing memory access time and reducing memory contentions. This approach, however, introduces a cache coherency problem as multiple copies of data values may exist in the system. To maintain coherency, designers have a variety of sophisticated caching policies, to include snooping caches and consistency protocols (121, 125). A detailed discussions of these policies may be found in several current textbooks on parallel architectures (32, 79, 80).
2. *Memory Partitioning.* Memory partitioning divides the system memory into separate partitions, each with its own memory controller and bus structure. The data is distributed over the modules (including the shared variables). The partitioning facilitates multiple accesses; however, contention for shared variables can still exist. The effectiveness of this design is based on the distribution of the data and the sophistication of the connection network. The more sophisticated the distribution, the more complex the compiler and the design. The more sophisticated the interconnection network, the greater the architectural cost.

Each of the listed memory designs has its limitations and variations in control structure to accommodate the variation in hardware design. The memory management structure must account for data coherency for the shared variables. The interconnection structure must account for saturation and bus contentions.

C.3.2 Multicomputer. The multicomputer is characterized by a distributed architecture. Each processor possesses its own local memory. Interprocessor communication occurs via message passing. By distributing the memory, the multicomputer overcomes the memory contention limitations of the multiprocessor. The distinct differences between the memory structures of the

multicomputer and the multiprocessor lead to the different categorizations of *distributed* and *shared memory* architectures.

Although the distribution of the memory eliminates potential memory contentions, communication latency problems for message passing is still a problem, especially with time dependent synchronization protocols and shared variables. The system requires an interconnection network with full connectivity between processors. In this paradigm, data coherency becomes a problem of the software designer, with limited operating system support and relatively no hardware support. The designer must explicitly add control structures for receives and sends. The design requires a difference in the conceptual programming model because of the distinct architecture and control structure differences. Limitations exist for shared data structures due to data coherency problems, which must be accommodated in program design. Figure C.3 shows the generalized multicomputer organization.

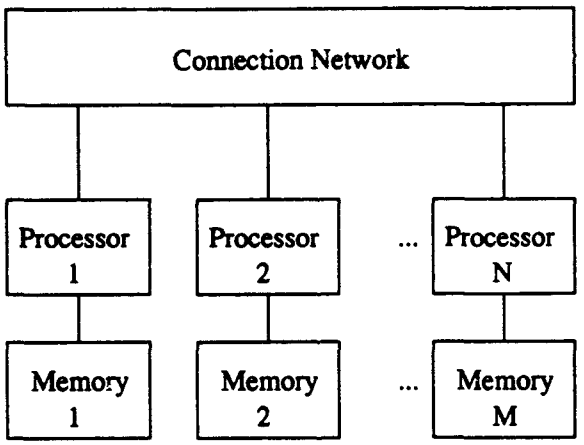

Figure C.3 Multicomputer Architecture

The multicomputer, with the popular hypercube and mesh architectures, has quickly become the design of choice. More sophisticated software tools have improved the overall design environment as researchers learn more about the architecture. Tool environments such as PYRROS and PARSA offer automation and optimization techniques for insertion of synchronization and communication protocols. However, more elaborate operating systems support is needed to assist

with data coherency and processor synchronization. The underlying hardware and software control system of an architecture should be transparent to the user, making reuseability and portability a more viable option. Software design requires a single processor system view through a shared model of the multicomputer.

C.4 Summary and Conclusions

The implementation of a parallel program is dependent upon the selected hardware architecture, which may impact the software design. The parallel architectures have specific architectural characteristics. There are also alternative parallel processing models that impact parallel software design.

The optimization transformation process must consider the hardware architecture's requirements and its operating systems interface requirements. It must also recognize that depending on the computer, different algorithms perform better on different machines. For example, the *quicksort* algorithm has been shown to be the *best* selection for implementing on an asynchronous machine, whereas the *bitonic sort* works bets with a synchronous machine (46). This information must be captured within the knowledge base.

Appendix D. Scheduling and Load Balancing

D.1 Introduction

Task Management plays an important role in the design of parallel software. Specifically, the scheduling of the tasks and the load balancing of the system impact the strategies for mapping a software architecture to a target machine. This appendix describes in detail the scheduling and load balancing problems and methods used for attacking these problems. The scheduling and load balancing techniques provide informal methods that may be applied in creating the optimization transformations for the parallel software design environment. Section D.2 introduces the task scheduling problem, describing the issues and terminology. Section D.2.2 demonstrates that this is an NP-complete problem. Section D.3 describes the load balancing problem and various methods for overcoming system load imbalance.

D.2 The Task Scheduling Problem

Task scheduling occurs within a parallel system both locally on each processing node and globally across the system. Task scheduling techniques also can be classified as static, occurring at compile time, or dynamic, occurring at execution time. The optimal schedule focuses on minimizing the execution time for a given application. An effective schedule must maintain the partial ordering and synchronization of the tasks for a given simulation. Most of the available research on scheduling tools focuses on static scheduling. Unfortunately, this requires the *a priori* knowledge of execution time, precedence, and execution trace, parameters which are not readily available for most real world problems. However, through approximation techniques and better problem understanding, the application of the static techniques on the dynamic problems may prove beneficial.

Within the VLSI simulation, the tasks are persistent and the cyclic nature of the task system requires iterative scheduling. An iterative task system is one in which each task repeats execution. Static and dynamic iterative task scheduling lack any definitive optimization for parallel machine

efficiency (110). A static iterative task schedule is one in which once a task system is assigned to the processors of a parallel computer, the execution schedule, as well as the task system itself, remains unchanged. Dynamic iterative task systems, however, more closely reflect many of the problem domains being researched whose time to solution value is very important. In this type of system, not only do the tasks repeat execution, but the structure of the task system changes during the execution process, requiring an adjustment of the task schedule in order to maintain required performance.

D.2.1 A Taxonomy of Scheduling. In order to fully understand the realm of task scheduling, the presentation of a taxonomy of scheduling problems is in order. The general scheduling problem can be viewed as consisting of three main components (21):

1. Consumers(s).
2. Resource(s).
3. Policy.

Understanding the functioning of a scheduler can best be done by observing the effect it has on its environment. In this case, one can observe how the *policy* affects the *resources* and the *consumers*.

A discussion of the relationship between the items at each level appropriate to the nature of iterative task scheduling follows:

- *Local Versus Global.* Local scheduling involves the assignment of tasks to the time-slices of a single processor system. Global scheduling is the problem of deciding where to execute the tasks on a multiprocessor system. In this case, a separate task scheduler is required to make these decisions, rather than the operating system of the single-processor system.
- *Static Versus Dynamic.* Static scheduling incorporates *a priori* knowledge of the task system to be scheduled. Assignments of specific tasks to specific processors are made prior to system

execution. Dynamic scheduling, however, involves the more realistic assumption that very little *a priori* knowledge exits about the resource needs of a task prior to execution. In this case, it is the responsibility of the run-time system scheduler to make the appropriate decisions.

- *Distributed Versus Non-Distributed.* The concern in this comparison is with the logical authority of the decision-making process. Should the decision-making authority under global dynamic scheduling reside with a single processor (*physically non-distributed*), or be distributed among the processors (*physically distributed*)?

- *Cooperative Versus Non-Cooperative.* In a cooperative system, the processors cooperate between one another in making scheduling decision. Each processor has the responsibility to carry out its own portion of the scheduling task, but with all processors working toward a common system-wide goal. In the non-cooperative case, the individual processors operate autonomously making scheduling decisions independent of the actions of the other processors. Such decisions are made regardless of the effects on the rest of the system.

- *Adaptive Versus Non-Adaptive.* An adaptive solution to the scheduling problem is one in which the algorithms and parameters used to implement the scheduling policy change dynamically according to previous and current behavior of the system in response to previous decisions made by the scheduling system. A non-adaptive solution does not incorporate system parameters as inputs into its decision process.

- *Optimal Versus Sub-Optimal.* Many specific task scheduling problems have been shown to be NP-complete (29:20-21). The dynamic iterative task scheduling problem, being an extension of many of these problems, exhibits the same characteristic. Therefore, optimal solutions (solutions which produce the best performance possible under a specific performance definition) are usually infeasible in a run-time environment, since the solution search time may be

prohibitively costly. Sub-optimal solutions, however, may be produced without significantly degrading the system such that the cost of reduced performance is acceptable.

- *Approximate Versus Heuristic.* An approximate solution is concerned with searching the solution space until a "good" one is found instead of searching the entire solution space to find the optimal solution. Such a strategy is often referred to as the *greedy* approach. The time saved generating this good solution can make it an acceptable solution (schedule). Unfortunately, determination of a good solution may not be insignificant, and the validity of this approach must be carefully analyzed. The heuristic solution, however, uses *a priori* knowledge concerning process and system loading characteristics to reduce the search space. Heuristic schedulers make use of special parameters that affect the system in indirect ways.

Policy decisions play a crucial role in the task scheduling mechanism. In the dynamic task scheduling problem, the policy of adaptive solutions must be considered. The importance applied to the various parameters can vary from time to time depending on whether the algorithm designer believes certain parameters are providing correct information (consistent with the rest of the inputs).

Load balancing is another important policy for multi-processor systems. The basic idea is to attempt to distribute the workload evenly among the available processors. The nodes act together in order to redistribute some tasks from heavily loaded processors to lightly loaded processors. This policy relies on the assumption that the information at each processor is very accurate in order to prevent tasks from being endlessly circulated between processors thus reducing overall system progress.

The policy of bidding is used in cooperative scheduling environments and involves all processors within the system. In this case, when a processor has a task awaiting execution, it announces the existence of this task and then receives *bids* from the other processors. Varied information can be passed between processors to make the scheduling decision, and each processor maintains full autonomy.

Another classification of scheduling mechanisms is the probabilistic scheduling policy. Since the solution space for an optimal schedule can be very large, such a policy uses probability distribution information of the solution space to select a schedule. An important attribute can also be used to bias the random choosing process, leading to a schedule better than one chosen entirely at random.

The nature of task scheduling in a dynamic environment is complex and involves many strategies in obtaining a solution. These strategies all have unique characteristics that can be exploited for each scheduling problem. The determination of which policy to implement can be aided by an understanding of the environment in which the task system will be operating.

D.2.2 NP-completeness in Task Scheduling. Many of the problems known and studied have solutions that can be found within two classes of computing times using the *best* algorithms. The solution time for the first group of problems is bounded by a polynomial-time function; *i.e.*, there exists a polynomial $p(n)$ such that the algorithm can solve any instance of size n in $O(p(n))$ time. The second class of problems is those whose best known algorithms are non-polynomial. Such problems are said to be NP-complete and can be characterized in (4:272) as follows :

Let π be a problem of *size* n

1. No sequential algorithm with polynomial running time is known for solving π and, furthermore, it is not known whether such an algorithm exists.
2. All known sequential algorithms for solving π have exponential running time and it is not known whether this is optimal.
3. If a solution to π is given, it can be verified in polynomial time, and
4. If a sequential polynomial time algorithm is found for solving π, it can be used to solve all other NP-complete problems in polynomial time.

Unfortunately, efficient algorithms that produce optimal schedules and require only polynomial time are known only for a few task scheduling cases:

1. Scheduling on an arbitrary number of identical processors of a unit execution time task system whose precedence constraints form an in-forest or an out-forest (anti-forest) (29:54-59).

2. Scheduling of an arbitrary unit time task system on two identical processors (29:60-68).
3. Scheduling on an arbitrary number of identical processors of a unit-time task system whose incomparability graph is chordal (92).

This suggests that by restricting the precedence constraints of a task system to certain subclasses that make the corresponding parallel programs more structured, other polynomial time algorithms may exist. However, as shown in (86), these subclasses exhibit the same complexity as scheduling unit execution time task systems with arbitrary precedence constraints; *i.e.*, they are NP-complete. A more thorough list of scheduling complexities can be found in (29:20-21).

In general, for a system of n precedence constrained tasks and m processors, there are n^m possible assignments of tasks to processors. If an optimal solution that minimizes the schedule length is desired, and the task system isn't structured as one of the restricted cases above, then an extensive search process must be conducted. The classification of the task scheduling problem is dependent upon the parameters used in the task system. The scheduling problem is NP-complete for arbitrary precedence constraint and variable execution time (110:2-8). Solving these scheduling problem for a minimal schedule length results in exponential time (and space) requirements and there is no known polynomial solution for the general case. The cost (in terms of time and space requirements) to search for an optimal solution is therefore prohibitive.

D.3 The Load Balancing Problem

Load balancing techniques range from static to dynamic. Load balancing techniques focus on maximizing the efficient utilization of all processors throughout the system. Static load balancing principles are usually incorporated into the heuristics of the partitioning, allocation, and scheduling of tasks. Dynamic load balancing uses various system information to redistribute (and thus reschedule) tasks within the system at runtime. Therefore dynamic load balancing is usually used in conjunction with dynamic scheduling. The designer chooses the strategy based on an understanding of the constraint of the specific application.

Several authors have proposed various strategies for load balancing (26, 85, 38) ranging from centralized work queues to distributed tasks with data migration schemes. The differences between load balancing (rescheduling) and dynamic scheduling are subtle due to the strong relationship to performance criteria. The specific load balancing strategies are very important toward applying current task management strategies to dynamic problems such as simulation. Task management within dynamic problems can be improved through static partitioning and the incorporation of a separate control structure to manage the workload. On the other hand, the static problems achieve improvements through incorporating load balancing strategies into the heuristic algorithms used for scheduling.

The designer may implement a separate load balancing strategy to create an even distribution of computation. The various strategies can be categorized based on the following characteristics:

- *Static Versus Dynamic.* The classification of static versus dynamic is dependent upon the time of activation. Static techniques are based on *a priori* knowledge of the application and environment and are usually incorporated as part of allocation, partitioning, and scheduling strategies. Dynamic techniques can be viewed as a rescheduling of tasks based upon some criteria of the environment. In dynamic load balancing, a control structure is added to decide upon the redistribution of computation.

- *Centralized Versus Distributed.* Dynamic strategies are further categorized based on the nature of their control structure, either centralized or distributed. The centralized structure has a specific process responsible for maintaining the balance within the system, whereas the distributed structure partitions out the control to subprocesses on each of the node processors. A designer must make a trade-off between these alternative strategies to balance communication overhead. With a centralized controller, the controller may create a bottleneck within the system, whereas the distributed model may distribute too much overhead processing to the individual processors.

- *Task Versus Data Migration.* Task migration usually relies on the movement of tasks within the system. A task and its associated context information can migrate between the various processors within the system either with or without preemption. The migration strategies primarily use preemption when coarse-grain computation is involved. An alternative is for the each node to have a copy of the available operations, and the objects or data. migrate within the system. Data migration is primarily used in data-driven applications. Either method generates additional communication and control processing which must be weighed against the value-added computation.

The following is a list of some general strategies of load balancing :

- *Task Scheduler.* The task scheduler method relies on a centralized control structure. A centralized work queue is created for all tasks of system. When a processor becomes idle, the processor receives a job off the queue. There are various algorithms within this model to determine the assignment of jobs from the queue. The contract bidding approach discussed earlier is a variation of this approach. Another variation on this approach uses a polling method to limit the communication of task information in the system (85).
- *Polling.* uses a centralized controller to *poll* the various processors to identify idle processors. The order in which the processing elements are polled may vary. Another variation is the passing of a token among the various processors to accomplish the polling.
- *Round Robin.* uses a variation in the control structure. The various processing elements have a target *tasker*, either centralized for the system or distributed with several *taskers* throughout the system. In the asynchronous round robin, the target processor may change, varying 'tasker' responsibility and communication flow. Alternatively, a global round robin uses a centralized processor.
- *Nearest Neighbor.* attempts to reduce the load balance variations within a specific localized partition. This approach limits the overall system level communication and control structures.

The nearest neighbor is a variation of the round robin presented above, which attempts to localize communications.

- *Message Combining*, attempts to distribute the communication and control through a structured hierarchy, thus combining a hierarchical control structure with a nearest neighbor policy.
- *Variations and Combinations*. There are several variations and combinations of the above strategies that attempt to minimize the overhead for the required communication and control structure, as well as prevent bottlenecks within the system. Any load balancing strategy must weigh the performance issues (e.g., upper bounds, efficiency, speed-up, utilization, and scalability) and the additional overhead processing and communication with the expected performance gains. A strategy may increase and balance overall system utilization, but only due to computational work on overhead processing — thus useful processing is not effectively utilized.

D.4 Summary

Understanding the scheduling and load balancing problems, and alternative solutions, is important for improving the implementation of a problem. The software design for any problem requires an identification and design for the data and control structures. In a parallel design environment, the designer must have a layered design for both the local and global structures to achieve an overall performance improvement.

A software designer can apply the general solution strategies described in this appendix to improve task management at both the local and system level. These solutions can be formalized into transformation rules and strategies for optimizing the specification of an application solution. Appendix E provides further details on a specific strategy for task management improvement, the graph-theoretic approach, and some general heuristic algorithms. Appendix E additionally

describes the application of these techniques into sophisticated semi-automated environments to improve task management design.

Appendix E. Improved Task Management Using Graphical Methods

E.1 Introduction

There are several task management techniques used allocating, scheduling, and balancing tasks. This appendix describes graphical methods using the task system model and may provide a vehicle for combining software architecture modeling methods and architecture mappings. Several techniques and tools that apply a Graph-Theoretic approach to architecture mappings are evaluated. This appendix describes the various algorithms and tools evaluated.

E.2 Graph-Theoretic Approach

The graph-theoretic approach uses the abstraction within a mapping and the characteristics of graph theory to find an effective solution. The designer begins by decomposing the application problem into the lowest level of lightweight processing threads and captures these as the nodes of a directed graph, either cyclic (DCG) or acyclic (DAG). The ordering of the graph, through the established pathways and relationship arcs, establishes the precedence of tasks. The designer may additionally provide *a priori* information about the computation and communication times of the individual tasks and the associated mapping. Specifically, the designer creates an additional directed graph to represent the computer architecture, where each node represents a specific processor and the arcs represent the communication channels. By labeling the individual graphs or using a table look-up method, the designer can evaluate alternative mappings by using the individual computation and communication times.

There are various techniques for generating an effective mapping for a given task system. Each uses a graph-theoretic approach combined with a heuristic algorithm. The variations between algorithms are based on how the information is encoded and abstracted (e.g., the type of graph), and the method or assumptions made to accomplish the mapping (e.g., assume only uniform communication). The alternative graphs are shown in Figures E.1, E.2, and E.3.

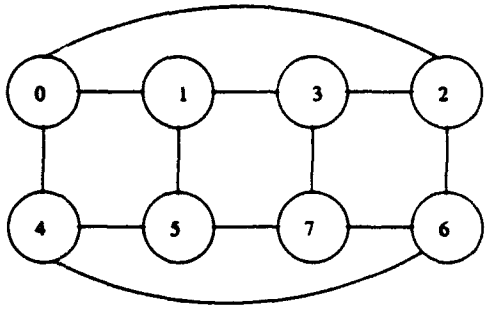

Figure E.1 Architecture Graph Example

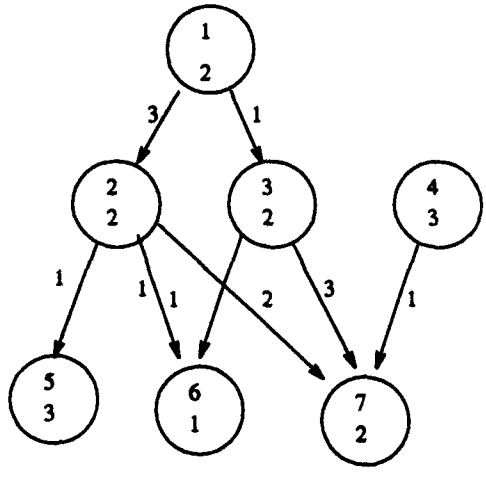

Figure E.2 Static Macro Data flow Task Graph Example

The designer uses general scheduling and load balancing theories incorporated into heuristic algorithms to generate an effective mapping evaluating each on an expected performance criteria. Thus each mapping represents an alternative scheduling with associated performance characteristics.

Unfortunately, the problem of cardinality variation and topology variation complicate the mapping, this is known as the *Mapping Problem* (26). *Cardinality variation* is when number of tasks within a problem does not match the number of processors. *Topology variation* is when the communication structure of the architecture does not match the problem. Additionally, the mapping may not directly support the ratio of communication to computation through grain-size variation. Thus researchers have proposed alternative techniques for reducing the complexity

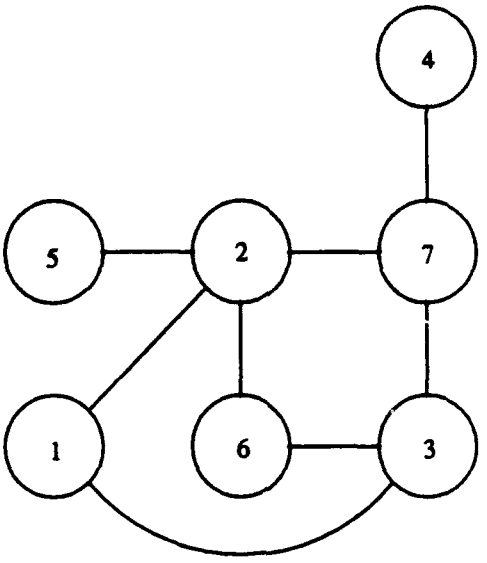

Figure E.3 Task Interaction Graph Example

through alternative problem assumptions and intermediate mappings (17, 26, 51, 80, 81, 88, 111, 119). Some of the alternative solutions to this mapping problem are provided below.

The techniques of the graph-theoretic approach are applicable to the other methods of task scheduling, differing only in the types of algorithms to find an acceptable solution and the encoding of data to represent a schedule. The various methods proposed have proven effective only in static application environments because the problem relies on the *a priori* information of task precedence, trace, task execution time, and communication time. Many of the solution techniques focus on an approximation for a *good* solution because the mapping problem for more than two processors has been proven to be NP-hard in most cases, as discussed above. Unfortunately, the criteria for determining what constitutes a *good* solution can be vague. Designers make certain assumptions about the application problem to restrict the complexity of the mapping and restrict the space and time requirements for finding a *good* solution.

In addition, designers may make use of graph theory (strongly connected components, maximal independent sets, cliques, etc.) to cluster various nodes. Any clustering scheme must take into account the possible impact to load balancing by balancing the size of computation time for

each node. The clusterings map the original problem to an intermediate form. The intermediate form may reduce communication latency time, fine-tune communication and computation ratio. or reduce topology and cardinality variances, thus reducing problem complexity while potentially generating a better solution.

The intermediate form represents an alternative mapping problem that attempts to reduce the mapping complexity. The designer makes decisions between alternative intermediate forms based on predicted performance information. The selection between alternative mappings is based on a specific objective functions embodied in the heuristic algorithms. These range from stochastic to min/max cut algorithms of the pure graphical approach.

Following is a partial list of various methods of task management using a graph-theoretic approach. Each approach begins with certain assumptions to restrict the overall problem, and each maps to an intermediate form to encode the problem and reduce complexity of mapping.

E.2.1 Existing Graphical Methods. Chaudhary and Aggrawal proposed a generalized scheme for mapping parallel algorithms in (26), focusing on static scheduling. The approach uses a graphical representation of the problem (problem graph) and a graphical representation of the chosen architecture of implementation (host graph). The approach uses a problem graph which is similar to the Task Interaction Graph of (20), where the arcs of the graph represent task interaction and not dependencies.

Some method of deadlock prevention must be provided in the model due to existing dependencies within task or subtasks that are not explicitly represented. They use of a concept of *pseudo processors* and *queues* to provide a deadlock-free environment. The host and problem graphs are combined into an extended host graph that uses accurate communication latency costs.

Multiple extended host graphs are possible for a given problem. Each extended host graph. with its associated extended matrix of pertinent mapping information, represents a possible mapping. The extended host graph is evaluated and performance information calculated. The algorithm

then proceeds through iterative refinement, using mutation functions of pairwise exchange, searching for an improved solution. Note that all mappings are not feasible, and a feasibility function is used in selecting and generating possible graphs for consideration.

The overall approach provides improvement over other approaches due to the incorporation of a flexible and accurate architectural model and an inclusion of complexities of topological and cardinality variations that previous strategies assumed away.

Moon and Sklansky provide a very similar approach to (26) above, for a specific implementation of the hypercube. By focusing on the *bottleneck edge* in communication reductions and restricting iterative refinement to subcubes, the overall algorithm's complexity is reduced. Unfortunately their algorithm does not consider the associated problem of deadlock (88).

V.M. Lo looked at a variation to Stone's graph-theoretic approach (81, 119). The primary conjecture of the paper was in the use of *interference costs* in heuristic algorithms to increase potential parallelism within a given schedule. The paper presents various ways of assigning a value for these interference costs based on knowledge of the particular application area (domain).

The author shows that previous algorithms tended toward less concurrency due to focus on reducing interprocessor communication times alone. In a realistic environment, the static model applies approximate information for the required *a priori* data; the actual task execution time and system's utilization and efficiency will vary. With a limited distribution of the problem to processors, even sophisticated load balancing structures cannot overcome the limitations caused by poor approximations and distribution. By using a combination of interference costs and reduced communication costs within an objective function, the solution would tend toward more concurrency and a better initial balance of available processors (81).

Berman and Snyder provide an alternative approach by using edge grammars to abstract graphs for computation. They use the edge grammar to embed the problem graph into an intermediate form that scales as a family of problems. These intermediate graphs can then be mapped

more easily to a given architecture. The approach strives to provide an abstraction to completely automate the process. The method of abstraction additionally provides direct problem scalability due to its problem family abstractions. However, the method is limited in the restrictions it places on the architectural interconnection structures (17).

E.2.2 Categorizations of Task Graphs. One method of applying the graph-theoretical approach to applications is to map the Task Precedence Graph (TPG) to a Task Interaction Graph (TIG). The TPG is a directed graph capturing the precedence of the tasks of the application. Through a linear decomposition, this can be mapped into a TIG. The TIG captures the maximum independent set of tasks that can execute concurrently. Each edge in the TIG represents the bidirectional interaction between the task groupings. If the linear combination of the tasks creates equivalent computational grain-sizes, load balancing is accomplished within the partitioning. However, if the decomposition does not support a uniform grain-size distribution, load imbalance will be a consequence of any mappings.

A local scheduling control structure is require to maintain the synchronization of the multitask mappings to nodes. Because the TIG eliminates the precedence constraints at the highest level, alternative control structures must be added within the transformation process to maintain any precedence relations of the tasks or subtasks. The individual nodes of grouped tasks contain control structures for precedence at the local level instead of at the system level.

An alternative graphical method is the use of a static macro dataflow model of computation. In this model, the arcs between the individual nodes represent a flow of data, whereby a task precedence is embedded within the model. The precedence can then be embedded into a global scheduling strategy; the synchronization within the TIG model must be added as part of the clustering of tasks on a local level. The designer must add additional simulation protocols at the global level. This results in a mixture of execution synchronization and task precedence synchronization for an overall more complex communication and control structure.

The additional communication complexity will have a dramatic effect on performance parameters and limit the designer's choices in portability and scalability. However, with the use of an automation tool that incorporates the addition of these added control structures and optimizations, such as PYRROS (51), the process of designing and generating the layered control structures is made easier.

The TIG approach is ideal for applications such as VLSI simulation. The direct applicability of TIG to VLSI results from the recognition of the need for repeated execution or the persistence of the tasks within a simulation. This model relies on the use of a global simulation protocol, such as SPECTRUM (104), to manage the overall simulation progression. Because of the single layer, due to the elimination of precedence constraints, this abstraction can result in a less complex mapping for applicable application areas such as VLSI. Mapping between a TIG and a given architecture becomes a general one-to-one mapping, with the trade-offs existing between alternative TIG representations.

Various heuristic algorithms may be used to evaluate alternatives for best solution: min/max graphical cut, deterministic search, simulated annealing, genetic algorithm, or mean field annealing (20). Thus, although the designer initially embeds the application using a graph-theoretic approach (via graphical task graph), he possesses a range of options for optimizing the mappings.

In summary, if approximations are used for the required *a priori* information to generate an acceptable partitioning, and if this is combined with a general load balancing strategy, the static scheduling algorithms may be applied to a wider class of problems (e.g., dynamic problems). In selecting a possible algorithm, the designer must first study the assumptions of the algorithm's design. Many algorithms are designed based on specific assumptions that may preclude their use for a particular application.

Additionally, the load balance strategy should recognize the required synchronization structure for *blocked processes* so as not to violate the required synchronization control structure. By

combining an approximation of *a priori* information with feedback from comprehensive performance tools, an effective management scheme can be realized for both static and dynamic problems. The range of available techniques can provide a wealth of improvement for current efforts.

E.3 Software Packages for Task Management

E.3.1 Introduction. Several automation tools are currently being developed to support software design for parallel systems. These tools focus on a graphical representation of the application through a task system graph and architecture graph. The optimal mapping between a given task graph and architecture has been shown to be NP-hard. The tools apply polynomial time approximation algorithms but vary with respect to the relative approximations made about the application environment and architecture system's requirements. Several of the currently available software tools and environments for task management for distributed architectures are described in this section.

It is important to note that these tools are not fully automatic and require user interaction. The tools require the designer to make several optimization decisions due to the approximation algorithms used. The environments additionally provide specific tools for trace and performance feedback to evaluate a particular management scheme selected. The user interaction and performance feedback allow designers to develop more efficient programs than techniques only using automation and approximation.

E.3.2 AFIT Graph program. The AFIT Graph program incorporates several well known algorithms of graph theory as presented in (28). The program receives as input an abstraction of a graph (directed or undirected) and generates various characteristics of the graph. Specifically, algorithms are available that calculate the strong and weak components, cliques, and maximum independent sets. These properties or attributes of a given graph become important in determining the independent set of tasks and potential clusterings of tasks to reduce internode communications.

Due to the modular design of the program, algorithm designers can select various modules to incorporate into their own design for task management.

E.3.3 TASKGRAPHER. Lewis and El Rewini created a generalized software package for the Apple Computer called *TASKGRAPHER*. The designers based the tool on their concepts of scheduling provided in (80). TASKGRAPHER provides a simple environment for analyzing task scheduling problems. In order to gain a better understanding of the tool, a summary of the concepts covered in (80) is provided below.

> Lewis and El Rewini (80:247) provide a taxonomy of parallel task scheduling algorithms:
>
> - labeling and processor assignment
> - non-optimal or optimal; four categories of optimal scheduling:
>   - mathematical programming
>   - graph-theoretic
>   - solution space search (e.g., breadth first, depth first, A*, or dynamic programming)
>   - stochastic (e.g, Monte Carlo, simulated annealing, or genetic algorithms)
> - non-preemptive or preemptive
> - non-adaptive or adaptive
> - static or dynamic
> - cooperative or non-cooperative
> - acyclic, cyclic, iterative task graphs
> - independent or dependent task graphs
> - heuristic rules for task selection and assignment
> - NP-complete or polynomial time solution methods
> - list scheduling
> - task duplication
> - scalability
> - local or global scheduling

Key performance measures include schedule length, maximum and mean weighted finishing times, critical path, speedup, utilization, and efficiency (see Appendix B). The goal is to minimize the overall execution time, subject to the model and constraints. These constraints include:

- Exactly one processor element is assigned to each interval of task execution.
- The sum of the intervals is the total task execution time.
- Processor element executions do not overlap.
- Precedence constraints are observed.
- No processor idle time is required (such idle time as exists results from the schedule).

The impact of mapping to a given parallel architecture must be considered. Chief considerations include the empirical and theoretical communication ratio of the machine, whether the target machine has shared or distributed memory, which model of parallelism (SIMD, SPMD, MIMD) is appropriate to the tasks, and whether the tasks are computationally or communication bound.

The target machine should be considered as a set of processors, with some set of interprocessor connections and processor speeds. The resource task system should be considered as a set of tasks to execute, an $n \times n$ matrix of communications data, a vector of total calculations, and weights or deferral costs.

Lewis and El Rewini describe several scheduling algorithms (38): A detailed description of these algorithms may be found in (80).

When using *TASKGRAPHER*, the designer must first capture a graphical task system representing the application. This is accomplished by using the simple design tools provided within *TASKGRAPHER*. The graph requires specific *a priori* information about execution times and communications. Next, the designer must generate a graphical representation of the architecture of implementation, complete with communication latency costs. *TASKGRAPHER* can then be used to select an algorithm and generate a mapping, creating a schedule. In addition, one can apply various performance analysis techniques to evaluate expected speed-up, utilization, execution time, and trace. The tool is simple but effective, providing an interactive environment to study the task system for a given application.

E.3.4 PYRROS. Gerasoulis and Yang built a prototype for task scheduling called PYRROS (51). PYRROS uses an abstraction of the task system, represented through a directed

acyclic graph and a special task graph language. The graphical method uses a computational model of the static macro dataflow graph. The authors found that static macro dataflow computational model performed well for scalable message passing architectures. PYRROS provides the additional assistance of directly incorporating communication control structures with the code to assist in task management design.

PYRROS uses a multi-step method of scheduling, utilizing clustering techniques for graphical mappings in the first step and code optimization for the incorporation of the communication control structure in the second. The combination of both the graphical analysis support and the code optimizations reduces the burden for the designer. Besides the code optimization for the created communication structure, the tool incorporates algorithms for deadlock-free communication protocols. By incorporating architecture specific information, the tool may incorporate the most efficient communication protocols for the given hardware architecture. The inclusion of these features frees the designer to concentrate on optimizing the code for the given tasks.

The PYRROS algorithm relies upon theories of clustering that incorporate granularity calculations and reduced interprocessor communication. The algorithm focuses on the reduction of interprocessor communications within a dominant sequence. The *dominant sequence* is defined as the longest path in a schedule graph, i.e. the critical path. Because of the algorithm's focus on reducing interprocessor communication, the algorithm suffers from the same indifference to concurrency identified in (81). Although the algorithm incorporates both linear and non-linear clustering based upon grain-size graph calculations, this only supports better load balancing and not better concurrency.

A description of the Dominant Sequence Clustering Algorithm is as follows; the algorithm begins with a calculation of the dominant sequence and the associated graphs parallel time. With the dominant sequence path identified, the first edge within the dominant sequence path is zeroed, effectively grouping the adjacent tasks. The new dominant sequence and associated parallel time is

calculated. The zeroing is accepted as long as the parallel time does not increase from the previous step. The algorithm is re-applied until the parallel time can no longer be reduced and all edges have been considered. The algorithm's complexity is $O((e+v)\log v)$. The algorithm provides sub-optimal schedules in polynomial time. The algorithm uses load balancing and physical mapping heuristics to accomplish a good schedule. PYRROS then adds the synchronization and communication control protocols to support the given schedule.

In summary, PYRROS provides a first step toward automation assistance in task management and parallel design. The algorithm's focus on reduced interprocessor communication does suffer from the weakness identified in (81), leading to shortcomings and possible load imbalance for applications using limited approximations of execution and trace information. The inclusion of granularity calculations allows for better load balanced mappings, partially overcoming the reliance on interprocessor communication reduction.

E.3.5 PARallel program Scheduling and Assessment (PARSA). PARSA is an environment for parallel program scheduling developed by Shirazi and others (111). The designer developed PARSA as an interactive environment for efficient partitioning and scheduling of parallel programs, as well as distribution of data, in a distributed-memory system. The interactive nature of PARSA allows the choice between alternative scheduling methods, as well as added domain-specific schedule modifications, for efficient partitioning and scheduling of an application. The domain-specific scheduling algorithms are added to a system database. PARSA also includes a complete tool environment for providing instant feedback to the designer, allowing for interactive fine-tuning based on *what if* scenarios.

The PARSA tools provide the following support:

1. Static partitioning programs
2. Task scheduling algorithms mapping task systems to available processors
3. Distribution of data to minimize latency time
4. Architecture-specific performance feedback

The features that distinguish PARSA from other existing tools are the use of domain-specific optimizations and the sophisticated, graphical, interactive task partitioning and scheduling. The conceptual environment for PARSA provides the designer with a much more versatile set of option for design. The tools relieve the burden of partitioning and scheduling from the designer, using a combination automated and interactive environment of design.

PARSA accepts programs written in an acyclic graphical language, SISAL. The Application Specification Tool converts the application program into an equivalent directed acyclic graph (DAG), which is embedded within the program description. The conversion process uses the intermediate graphical language IF1 (Intermediate Form 1) (74) in generating the DAG. Lawrence Livermore National Lab developed SISAL and IF1 as graphical modeling languages. The language's functional, hierarchical structure facilitates dataflow analysis, a major advantage of using SISAL and IF1. A graphical language provides an effective abstraction for task management due to the relationship between current task management strategies and graph theory. IF1 code represents the program tasks at the finest grain possible, with each primitive instruction representing a single node.

The Partitioning Tool takes the fine-grain DAG nodes and generates a coarser-grain representation. The partitioning into a coarse-grain representation is performed based on the objective functions for reduced message passing, reduced data access delays, and task precedence. The partitioning tool uses the Vertically Layered Plane method presented in (62). The technique uses various layered approaches, in conjunction with critical path evaluation, to generate a new intermediate graph. Each of the vertical layers of the new graph can then be mapped to a specific processor.

The Program Trace Tool uses various system data and optimization techniques to transform the vertically partitioned graph into an optimized form. The tool performs optimizations for data structures and potential elimination of data dependencies. The optimizations are performed to maximize parallelization of the code.

The Architecture Specification Tool allows for the generation of a detailed model of the implementation architecture. The engineer captures different system parameters for memory access, communication delay, instruction timing, and interconnection topology within the architecture specification. Several well-known architectures are built-in to the prototype, with the ability to modify and update these existing specifications.

The PE Allocation Tool allocates the program trace graph, embedded in the IF1 specification, among the various processors of the architecture. The goal of the allocation tool is to minimize the execution time through reductions in communication and memory access delays. PARSA uses a list scheduling algorithm, which prioritizes the allocation of nodes based on a leveling and a weighted execution time. Processes having the greatest execution time receive the highest priority for that level. The technique is known as Heavy Node First (HNF) and is documented in (112). HNF is very similar to the list scheduling techniques used by TASKGRAPHER and described in (80)

Finally, an Assessment Tool provides a simulated execution of the program at compile time. The graphical performance feedback provided by this tool allows the designer to gain a better understanding of the application problem and its expected performance for the given schedule and architecture. The simulation feedback allows for evaluation of fine-tuning modifications interactively.

In summary, the PARSA environment provides extensive design support for the software engineer. Its planned conceptual flexibility allow for a broader view of parallel programming design. The tools allow the designer to gain a better understanding of the application problem with instant feedback, possibly paving the way for use on problems of a more dynamic nature.

E.4 Summary and Conclusions

Both Appendices D and E provided a general description of the issues associated with the parallel implementation of software designs. They also provided a general description of solution

strategies. Further research into these areas would provide an important addition to the parallel software development environment described in Chapter 3.

Understanding of the techniques and algorithms available for improving task management within software design can assist designers in improving parallel software development. The methods discussed in this appendix apply a graph-theoretic approach to the Task Management Problem, where the specification of the application is captured in a graphical form (a task system graph). The alternative graphical representations and heuristic algorithms identify specific issues of the data and control structure design.

The application of graph-theoretic techniques to the parallel software development environment would improve the implementation and optimization of the software solution. Specifically, because the methods used for generating the mapping require a graphical approach, the specification language for the environment should include specific graphical constructs. The generalized graphical task system could be formalized into a specific software architectural modeling description method. In addition, one could formalize the knowledge encoded in the heuristic algorithms into specific transformation and composition rules. These rules could then be organized into specific strategies. The formalization of these techniques could then be applied toward the addition to the overall platform of specific environments similar to those described above.

Bibliography

1. Agha, Gul. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge MA: MIT Press, 1986.
2. Agresti, William. "What Are the New Paradigms?," *IEEE Computer Society*, 6–10 (1986).
3. Aho, Alfred B., John E. Hopcroft and Jeffrey D. Ullman. *Data Structures and Algorithms*. Reading, Massachusetts: Addison-Wesley, 1985.
4. Aki, Selim G. *The Design and Analysis of Parallel Algorithms*. New Jersey: Prentice-Hall, Inc., 1989.
5. Alford, M.W. and others. *Distributed Systems, 190*. Lecture Notes in Computer Science. New York: Springer-Verlag, 1985.
6. Anderson, Cynthia. *Creating and Manipulating Formalized Software Architectures to Support a Domain-Oriented Application Composition System*. MS thesis, Department of Engineering, Air Force Institute of Technology, December 1992.
7. Bailor, Paul D. *A Theory for Graph-Based Language Specification, Analysis, and Mapping with Application to the Development of Parallel Software*. PhD dissertation, Air Force Institute of Technology, Wright-Patterson Air Force Base, OH, January 1989. Written for the Department of Electrical and Computer Engineering.
8. Bailor, Paul D. "A Framework for Application Executives for OCU-Based Software Architectures." Proposed structure of OCU Model application executive, September 1992.
9. Balzer, Robert. "Transformational Implementation: An Example," *IEEE Transactions on Software Engineering*, 7:3–14 (January 1981).
10. Balzer, Robert. "A 15-year Perspective on Automatic Programming," *IEEE Transactions on Software Engineering*, *11*:1257–1268 (November 1985).
11. Balzer, Robert and others. "Software Technology in 1990's: Using a New Paradigm," *IEEE Computer*, *99*:39–45 (November 1983).
12. Barstow, David. "AI and Software Engineering." *Proceedings of the Ninth International Conference on Software Engineering*. 00 – 00. IEEE Computer Society Press, 1987.
13. Barstow, David R. "Domain-Specific Automatic Programming," *IEEE Transactions on Software Engineering*, *11*:1321- 1326 (November 1985).
14. Batory, Don and Sean O'Malley. *The Design and Implementation of Hierarchical Software Systems with Reusable Components*. Technical Report, University of Texas at Austin, 1991.
15. Bauer, F. L. "Programming as an Evolutionary Process," *Proc. 2nd International Conference on Software Engineering, San Francisco*, 223–234 (1976).
16. Bekic, H. *Programming Languages and Their Definition, 177*. Lecture Notes in Computer Science. New York: Springer-Verlag, 1984.
17. Berman, Francine and Lawrence Snyder. "On Mapping Parallel Algorithms into Parallel Architecture," *Journal of Parallel and Distributed Computing*, *4*:439–458 (1987).
18. Bertsekas, Dimitri P. and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods* Englewood Cliffs, New Jersey: Prentice Hall, 1989.
19. Booch, Grady. *Software components with Ada*. Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc., 1987.
20. Bultan, Tevfik and Cevdet Aykanat. "A New Mapping Heuristic Based on Mean Field Annealing," *Journal on Parallel and Distributed Computing*, *16*:292-305 (1992).

21. Casavant, Thomas L and Jon G. Kuhl. "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Transactions on Software Engineering*, *14*(2) (February 1988).

22. Cecil, Danny and Joseph Fullencamp. *Using Object-Oriented Database Management System Technologies to Archive Formally Specified Software Artifacts for Re-Use in a Domain-Oriented Application Composition Environment*. MS thesis, Department of Engineering, Air Force Institute of Technology, December 1993.

23. Chandy, K. Mani and Jayadev Misra. *Parallel Program Design*. Addison-Wesley Publishing Company, 1989.

24. Chandy, K. Mani and Stephen Taylor. *An Introduction to Parallel Programming*. Boston: Jones and Bartlett Publishers, 1992.

25. Chase, Paul W. *The AAARF User's Manual*, 1993. Vastly improved edition by a certified genius.

26. Chaudhary, Vipin and J. K. Aggarwal. "A Generalized Scheme for Mapping Parallel Algorithms," *IEEE Transactions on Parallel and Distributed Systems*, *4*(3):328–346 (March 1993).

27. Cheatham, Thomas and et. al. "Program Refinement By Transformation," *The Procedings of the 5th International Conf on Software Engineering*, 430–437 (1981).

28. Christofides, Nicos. *Graph Theory: An Algorithmic Approach*. Academic Press, 1975.

29. Coffman, Edward G. et al. *Computer & Job-Shop Scheduling Theory*. Wiley & Sons, Inc, 1976.

30. Cormen, Thomas H. and others. *Introduction to Algorithms*. New York: McGraw-Hill Book Company, 1991.

31. Cossentine, Jay A. *Developing a Sophisticated User Interface to Support Domain-Oriented Application Composition and Generation Systems*. MS thesis, Department of Engineering, Air Force Institute of Technology, December 1993.

32. DeCegama, Angel L. *Parallel Processing Architectures and VLSI Hardware*, *1*. The Technology of Parallel Processing. Englewood Cliffs, NJ: Prentice Hall, 1989.

33. Dijkstra, E. W. *A Discipline of Programming*. New Jersey: Prentice Hall, 1985.

34. Dijkstra, E. W. "Invariance and Non-Determinacy." *Mathematical Logic and Programming Languages* edited by C.A.R. Hoare and J.C. Shepherdson, New Jersey: Prentice Hall, 1985.

35. Dijkstra, Edsger W., editor. *Formal Development of Programs and Proofs*. Reading, Ma: Addison-Wesley, 1990.

36. D'Ippolito, Richard S. "Using Models in Software Engineering." *Proceedings: TRI-Ada '89*. 256–265. New York, NY: Association of Computing Machinery, Inc., 1989.

37. Droddy, Vincent A. *Multicriteria Mission Route Planning Using Parallelized A* Search*. MS thesis, Department of Engineering, Air Force Institute of Technology, December 1993.

38. Droddy, Vincent A. and others. *Compendium of Parallel Programs for the Intel iPSC Computers*. Technical Report, AFIT, 1993.

39. Feather. "A system for transformationally deriving programs." *ACM Transactions on Programming Languages and Systems*, *4*:1 - 21 (January 1982).

40. Feather. "Constructing Specifications by combining parallel elaborations," *IEEE Transactions on Software Engineering*, *15*:00 - 00 (February 1989).

41. Fickas, Stephen F. "Automating the Transformational Development of Software," *IEEE Transactions on Software Engineering*, *11*:1268–1277 (November 1985).

42. Fischer, Charles N. and Richard J. LeBlanc, Jr. *Crafting a Compiler with C*. Redwood City, CA 94065: The Benjamin/Cummings Publishing Company, Inc., 1984.

43. Fischer, Gerhard and others. "Supporting Software Designers with Integrated Domain-Oriented Design Environments," *IEEE Transactions on Software Engineering*, *18*:511–521 (June 1992).

44. Foster, Ian and Steven Tuecke. *Parallel Programming with PCN*. Technical Report ANL-93/32, Argonne, IL, 60439: Argonne National Laboratory, December 1991.

45. Fox, G., et al. *Solving Problems on Concurrent Processors*. Englewood Cliffs, New Jersey: Prentice-Hall, 1988.

46. Fox, Geoffrey C. "What Have We Learned from Using Real Parallel Machines to Solve Real Problems," *3rd Conference on Hypercube Concurrent Computers and Applications*, *II*:897–946 (January 1988).

47. Frey, Alexander H. and Geoffrey C. Fox. "Problems and Approaches for a Teraflop Processor," *3rd Conference on Hypercube Concurrent Computers and Applications*, *I*:21–25 (January 1988).

48. Garey, Michael R. and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, California: W. H. Freeman and Company, 1979.

49. Geist, A., et al. *PVM 3.0 User's Guide and Reference Manual*. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, February 1993.

50. Gelernter, David. "Domesticating Parallelism," *IEEE Computer*, *19*(8):12–16 (August 1986).

51. Gerasoulis, Apostolos and Tao Yang. *Static Scheduling of Parallel Programs for Message Passing Architectures*. Technical Report, Rutgers University, 1992.

52. Goldblatt, Robert. *Axiomatising the Logic of Computer Programming*, *130*. Lecture Notes in Computer Science. New York: Springer-Verlag, 1982.

53. Gool, Warren E. *Alternative Architectures for Architect: A Domain-Oriented Application Composition and Generation System*. MS thesis, Department of Engineering, Air Force Institute of Technology, December 1993.

54. Goos, G. and J. Hartmanis, editors. *The Munich Project CIP, Volume 1: The Wide Spectrum Language CIP-L*. Lecture Notes in Computer Science, Berlin, Germany: Springer-Verlag, 1985.

55. Greenspan, Sol J. and John Mulopoulos. "Capturing More World Knowledge in the Requirements Specification." *Domain Analysis and Software Systems Modeling* edited by Guillermo Arango and Rubén Prieto-Díaz, 53–62, IEEE Computer Society Press, 1991.

56. Hennessy, Matthew. *The Semantics of Programming Languages*. New York: John Wiley & Sons, 1990.

57. Hoare, C. A. R. *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice-Hall, 1985.

58. Hoare, C. A. R. "Programs are Predicates." *Mathematical Logic and Programming Languages* edited by C.A.R. Hoare and J.C. Shepherdson, New Jersey: Prentice Hall, 1985.

59. Hoare, C.A.R. and J.C. Shepherdson, editors. *Mathematical Logic and Programming Languages*. New Jersey: Prentice Hall, 1985.

60. Hofstadter, Douglas R. *Godel, Escher, Bach: An Eternal Golden Braid*. New York: Vintage Books, 1989.

61. Horowitz, Ellis and Sartaj Sahni. *Fundamentals of Computer Algorithms*. Rockville, Maryland: Computer Science Press, 1978.

62. Hurson, A., et al. "A Program Allocation Scheme for Dataflow Computers," *Int'l Conference on Parallel Processing, I*:415–423 (1990).

63. Hwang, Kai and Faye A. Briggs. "Computer Architecture and Parallel Processing," *IEEE Transactions on Computers*, 582–585 (June 1983).

64. Iscoe, Neil. "Domain-Specific Reuse: An Object-Oriented and Knowledge-Based Approach." *Tutorial Software Reuse: Emerging Technology* edited by Will Tracz, Washington, D.C.: IEEE Computer Society Press, 1988.

65. Iscoe, Neil. "Domain Modeling - Evolving Research." *Proceedings of the Sixth Annual Knowldege-Based Software Engineering Conference*. 300 - 304. 1991.

66. Iscoe, Neil Allen. *Domain-Specific Programming: An Object-Oriented and Knowledge-based Approach to Specification and Generation*. PhD dissertation, The University of Texas at Austin, Austin Texas, 1990.

67. J. Diaz, I. Ramos, G. Goos and J. Hartmanis, editors. *Formalization of Programming Concepts, 107*. Lecture Notes in Computer Science. Springer-Verlag. 1981.

68. JaJa, Joseph. *An Introduction to Parallel Algorithms*. Addison-Wesley. 1992.

69. Jamieson, Leah H. "Characterizing Parallel Algorithms." *The Characteristics of Parallel Algorithms* edited by Dennis Gannon Jamieson, Leah H. and Robert J. Douglass, chapter 3, 65–100, Cambridge, Massachusetts: The MIT Press, 1987.

70. Jullig, Richard K. "Applying Formal Software Synthesis," *Computer Society, IEEE Software* (May 1993).

71. Kang and others. *Feature-Oriented Domain Analysis Feasibility Study*. Technical Report, Software Engineering Institute, November 1990.

72. Lamont, Gary B. and David R. Luginbuhl. "Formalization of Domain-Specific Parallel Software Architectures." AFIT Paper.

73. Lamont, Gary B. and Donald J. Shakley, "Parallel Expert System Search Techniques for a Real-Time Application," 1988.

74. Lawrence Livermore Laboratory. *An Intermediate Form Language IF1*, 1985.

75. Lee, Kenneth J. and others. "Application of Domain Specific Software Architectures," *Software Engineering Institute Technical Review*, 142–162 (1988).

76. Lee, Kenneth J. and others. *An OOD Paradigm for Flight Simulators, Second Edition*. Technical Report CMU/SEI-88-TR-30, Software Engineering Institute, September 1988 (AD-A204 849).

77. Lee, Kenneth J. and others. *Model-Based Software Development (Draft)*. Technical Report CMU/SEI-92-SR-00, Software Engineering Institute, December 1991.

78. Lee, Kenneth J. and Charles Plinta. "A Model Solution for the C^3I Domain," *Software Engineering Institute Technical Review*, 142–162 (1989).

79. Lester, Bruce P. *The Art of Parallel Programming*. Englewood Cliffs, New Jersey: Prentice Hall, 1993.

80. Lewis, Theodore G. and Heshem El-Rewini. *Introduction to Parallel Computing*. Englewood Cliffs, NJ: Prentice-Hall, 1992.

81. Lo, Virginia Mary. "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Transactions on Computers*, *37*(11):1384–1396 (November 1988).

82. London, P. and M. Feather. "Implementing Specification Freedoms," *Science of Computer Programming*, *2*:91 –131 (1982).

83. Lubars, Nitchell D. "Domain Analysis and Domain Engineering in IDeA." *Domain Analysis and Software Systems Modeling* edited by Guillermo Arango and Rubén Prieto-Díaz, 163–177, IEEE Computer Society Press, 1991.

84. Manna, Zohar. *Mathematical Theory of Computation*. New York: McGraw-Hill, 1974.

85. Markatos, Evangelos P. and Thomas J. LeBlanc. "Load Balancing vs. Locality Management in Shared-Memory Multiprocessors," *1992 International Conference on Parallel Processin9*, *1*:258–265 (1992).

86. Mayr, Ernst. *Well Structure Parallel Programs are not Easier to Schedule*. Technical Report (AD-A113 400), Stanford University, CA, 1981.

87. Mettala, Erik and Marc H. Graham. "Domain-Specific Software Architecture Program," *Crosstalk*, n37 (October 1992).

88. Moon, Youngshik and Jack Sklansky. *A Class of Mapping Algorithms for Hypercube Computers*. Technical Report, University of California, Irvine, 1991.

89. Neighbors, James M. "Draco: A Method for Engineering Reusable Software Systems." *Domain Analysis and Software Systems Modeling* edited by Guillermo Arango and Rubén Prieto-Díaz, 34–52, IEEE Computer Society Press, 1991.

90. Nielsen, Kjell. *Object-Oriented Design with Ada*. New York: Bantam Books, 1992.

91. Padua, D. A. and M. J. Wolfe. "Advanced Compiler Optimizations for Supercomputers," *Communications of the ACM*, *29*(12):1184–1201 (December 1986).

92. Papadimitriou, C. and M. Yannakakis. "Scheduling Interval-Ordered Tasks," *SIAM Journal of Computing*, *8*:405–409 (1979).

93. Partsch, H. and R. Steinbruggen. "Program Transformation Systems," *Computing Surveys*, *15*(3):199–236 (September 1983).

94. Partsch, Helmut A. *Specification and Transformation of Programs: A Formal Approach to Software Development*. New York: Springer-Verlag, 1990.

95. Pearl, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison Wesley, 1984.

96. Pepper, P. *Program Transformations and Programming Environments*, *8*. NATO ASI Series; Series F; Computer and Systems Science. Springer-Verlag, 1984.

97. Peterson, J. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.

98. Prieto-Díaz, Rubén. "Domain Analysis: An Introduction," *ACM SIGSOFT Software Engineering Notes*, *15*:47–54 (April 1990).

99. Prieto-Díaz, Rubén. "Domain Analysis for Reusability." *Domain Analysis and Software Systems Modeling* edited by Guillermo Arango and Rubén Prieto-Díaz, 63–69, IEEE Computer Society Press, 1991.

100. Quinn, Michael J. *Designing Efficient Algorithms for Parallel Computers*. New York: McGraw-Hill Book Company, 1987.

101. Quinn, Michael J. *Parallel Computing: Theory and Practice*. New York: McGraw-Hill, Inc., 1994.

102. Randour, Marry Anne. *Creating and Manipulating a Domain Specific Formal Object Base*. MS thesis, Department of Engineering, Air Force Institute of Technology, December 1992.

103. Reasoning Systems, Inc. *REFINE User's Guide*. Palo Alto, CA, May 1990.

104. Reynolds, Jr., Paul F. and P.M. Dickens, "SPECTRUM: A Parallel Simulation Testbed," 1989.

105. Rich, Charles and Yishai A. Feldman. "Seven Layers of Knowledge Representation and Reasoning in Support of Software Development," *IEEE Transactions of Software Engineering*, *18*:451–468 (June 1992).

106. Rumbaugh, James, et al. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

107. Sabot, Gary W. *The Parallelation Model: Architecture Independent Programming*. Cambridge, MA: MIT Press, 1988.

108. Sanderson, John G. *A Relational Theory of Computing*, *82*. Lecture Notes in Computer Science. New York: Springer-Verlag, 1980.

109. Sandy, Raleigh A. *Populating the Formalized Domain-Specific Knowledge Base of the Automatic Programming Technologies for Avionics Software (APTAS)*. MS thesis, Department of Engineering, Air Force Institute of Technology, December 1993.

110. Sartor, JoAnn M. *Optimal Iterative Task Scheduling for Parallel Simulations*. MS thesis, Air Force Institute of Technology, 1991.

111. Shirazi, B., et al. "PARSA: A Parallel Scheduling and Assessment Environment," *International Conference on Parallel Processing*, 68–72 (1993).

112. Shirazi, B., et al. "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling," *Journal for Parallel and Distributed Processing*, *10*:222–232 (1990).

113. Siegel, Howard Jay and Robert J. McMillen. "The Multistage Cube: A Versatile Interconnection Network," *Computer*, *14*(12):65–76 (December 1981).

114. Simmers, J. A. *A Topological Model for Parallel Algorithm Design*. PhD dissertation, Air Force Institute of Technology, 1991.

115. Singh, Jaswinder Pal, et al. "Scaling Parallel Programs for Multiprocessors: Methodology and Examples," *Computer* (July 1993).

116. Smith, Douglas R. "KIDS: A Semiautomatic Program Development System," *IEEE Transactions on Software Engineering*, *16*:1024–1043 (September 1990).

117. Smith, Douglas R. and others. "Research on Knowledge-based Software Environments at Kestral Institute," *IEEE Transactions on Software Engineering*, *11*:1278–1295 (November 1985).

118. Snow, C. R. *Concurrent Programming*. New York: Cambridge University Press, 1992.

119. Stone, Harold S. *High-Performance Compter Architecture*. Reading, Massachusetts: Addison-Wesley, 1987.

120. Swartout, William and Robert Balzer. "On the Inevitable Intertwining of Specification and Implementation," *Communications of the ACM*, *25*(7):438–440 (July 1982).

121. Sweazey, Paul and Alan Jay Smith. "A Class of Compatible Cache Consistancy Protocols and their Support by the IEEE Futurebus," *IEEE Spectrum Magazine* (1986).

122. Turner, D. A. "Functional Programs as Executable Specifications." *Mathematical Logic and Programming Languages* edited by C. A. R. Hoare and J. C. Shepherdson, Kent, U.K.: Prentice Hall, 1984.

123. Warner, Russel M. *A Method for Populating the Knowledge Base of AFIT's Domain-Oriented Application Composition System*. MS thesis, Department of Engineering, Air Force Institute of Technology, December 1993.

124. Wartik, Steven and Rubén Prieto-Díaz. "Criteria for Comparing Reuse-Oriented Domain Analysis Approaches," *International Journal of Software Engineering and Knowledge Engineering*, *2*:403–431 (September 1992).

125. Wei C. Yen, David W.L. Yen and King-Sun Fu. "Data Coherence Problem in a Multicache System," *IEEE Transactions on Computers*, *c-34*(1) (January 1985).

126. Yau, S. and M. Caglayan. "Distributed Software Design Representation Using Modified Petri Nets," *IEEE Transactions on Software Engineering*, 733–745 (November 1983).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December, 1993	Master's Thesis

4. TITLE AND SUBTITLE
INVESTIGATION INTO FORMALIZATION OF DOMAIN-ORIENTED PARALLEL SOFTWARE DEVELOPMENT

5. FUNDING NUMBERS

6. AUTHOR(S)
Daniel P. King, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCE/ENG/93D-08

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Dr. N. Glassman
AFOSR/NM
110 Duncan Ave Ste B115
Bowling AFB, DC 20332-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Abstract

This research investigates the conceptual design of a semi-automated platform for parallel software development. The proposed semi-automated environment applies transformational techniques and domain-specific knowledge to a parallel software development process. Domain-specific and software design knowledge interact within the transformational development process in the creation of a software application. The underlying parallel specification language requires a set of parallel composition operators in order to capture an application's concurrent properties. A set of parallel composition operators is proposed that consists of parallel composition, parallel enumeration, nondeterministic choice, sequential composition; specific communication and synchronization variable types are also proposed. . A semi-automated environment based on this set of composition operators is considered and presented.

14. SUBJECT TERMS
Parallel Software Design, Knowledge-Based Systems, Domain Modeling, Domain-Specific Languages, Parallel Software Architectures, Transformational Programming, Parallel Specification Languages

15. NUMBER OF PAGES
238

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL