intellitech

The Intelligent Use of Technology

A SURVEY OF COMPUTER-AIDED ENGINEERING (CAE) TOOLS FOR THE DESIGN AND SIMULATION OF MULTIPROCESSOR SYSTEMS

A SURVEY OF COMPUTER-AIDED ENGINEERING (CAE) TOOLS FOR THE DESIGN AND SIMULATION OF MULTIPROCESSOR SYSTEMS

Report No. INT-82-15
March 1982

Authors: Dr. S.A. Mahmoud
Mr. J.G. Ouimet
Dr. C. Laferriere
Mr. W.T. Brown

Approved by: Dr. S.A. Mahmoud

Government of Canada / Gouvernement du Canada

Department of Communications

DOC CONTRACTOR REPORT DOC-CR-SP-82-045

DEPARTMENT OF COMMUNICATIONS - OTTAWA - CANADA

SPACE PROGRAM

TITLE: Survey of Computer-Aided Engineering (CAE) Tools For The Design And Simulation of Multiprocessor Systems

AUTHOR(S): S.A. Mahmoud
C. Laferriere
J. Ouimet
W. Brown

ISSUED BY CONTRACTOR AS REPORT NO: INT-82-15

PREPARED BY: Intellitech Canada Ltd.
352 MacLaren St.
Ottawa, Ontario
K2P 0M6

DEPARTMENT OF SUPPLY AND SERVICES CONTRACT NO: 3ER.36100-1-0273
SN: OER81-03151

DOC SCIENTIFIC AUTHORITY: R.A. Millar

CLASSIFICATION: Unclassified

DATE: March 1982

Preface

This work was performed for the Department of Communications, Communication Research Centre, under DSS Contract No. OER81-03151, entitled "Computer-Aided Engineering Tools for Spacecraft Multi-Microprocessor Design", from September 15, 1981 to March 31, 1982. This report is one of the following four contract deliverables:

1. Executive Summary

2. Report #1 - Review of Multiprocessor Systems and their Spacecraft Applications.

3. Report #2 - A Survey of Computer-Aided Engineering (CAE) Tools for the Design and Simulation of Multiprocessor Systems.

4. Report #3 - The Definition and Specification of an Integrated Set of CAE Tools for Spacecraft Multiprocessor System Design.

Acknowledgement

The study team gratefully acknowledges the technical guidance of Mr. R.A. Millar of the Communications Research Centre. His knowledge and experience in the field of computer simulation of spacecraft systems have contributed to the quality of the work and provided a constant source of encouragement to the study team.

As well, the study team wishes to thank Mr. J.M. Savoie also of C.R.C. for his fruitful discussions and critical reviews.

Table of Contents

List of Figures

1.0 Introduction

Interest in multiprocessor and distributed intelligence computer systems has increased dramatically in recent years. This interest has been fostered by the availability of microprocessors with ever increasing performance/price ratios and the expected emergence of monolithic systems with still higher capabilities in the near future.

The development of multiprocessor and distributed intelligence computer systems and their utilization in various applications have been impeded by the lack of an appropriate theoretical base. The control of systems containing more than several processors is not well understood. While considerable work has been done recently to develop a theoretical base, it seems unlikely that this work will have significant impact on practical system design in the near future. As a result, multiprocessor system designers have turned to the use of CAE tools for the development of such systems. Such CAE tools support the skill level of the designer, provide insight into the attributes of alternative architectures, allow evaluation of these architectures and support the development, simulation and test of actual multiprocessor systems.

The main objective of this report is to survey the multi-microprocessor computer aided engineering design tools that are currently available, as well as the design tools that are currently at various stages of research and development.

To understand the role, scope and utility of such multiprocessor design tools, it is worth reviewing the various design steps followed in a general top-down development process of a multiprocessor system. The first step involves the specifications of system requirements and is followed by

a description of the functional components of the system which are considered necessary to satisfy the requirements. This high level description of the functional components can then be translated into an intermediate design stage which involves the selection of a system architecture and its representation in the form of a system model. This model is used to describe the hardware and software structures and their relationships. Finally, at the lowest level, a register transfer model is used to provide details of the processing elements and to facilitate the hardware (logic) circuit design and test. In summary, the following design phases (see Figure 1.1) are identified in a top-down design approach:

(1) Specification of System Requirements

(2) System Function Specification

(3) System Architecture Selection

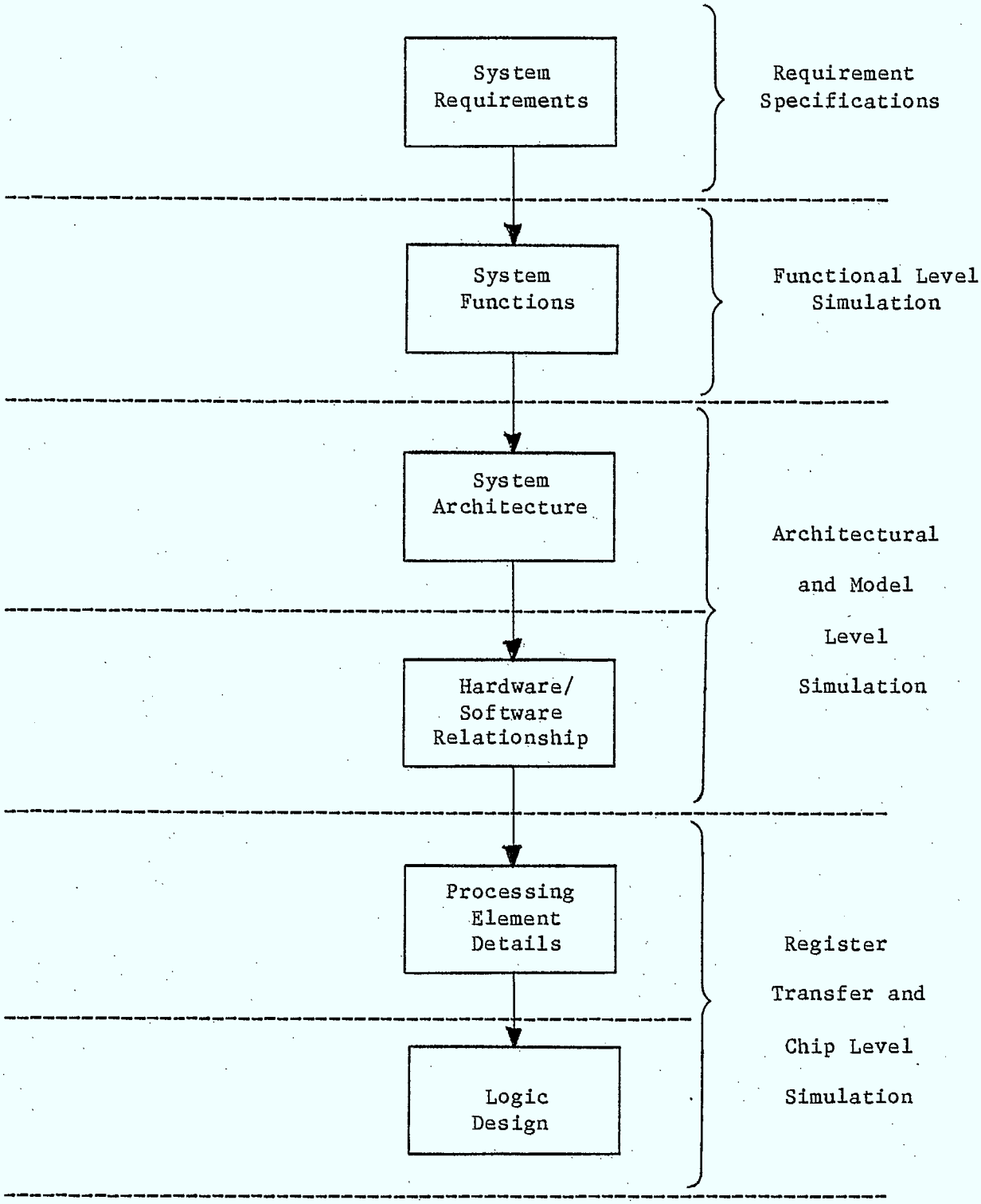
(4) Hardware/Software Relationship (System Model)

(5) Processing Element Details (Register Transfer Model)

(6) Logic Design

Before surveying CAE tools available to support each of the above specification/design levels, the issues examined at each level and the implications of following a top-down approach in the specification and design process are described.

Figure 1.1
Multiprocessor Specifications/Design Levels
and Corresponding CAE Tools



(1) Specification of System Requirements

The term "requirements" is used to describe any demand upon the system to be designed. A complete list of the requirements should be defined before the design process begins and documented in a requirement specification document. As this document constitutes part of a contract between the user and the designer, and between members of the design team, it should be consistent, unambiguous and non-redundant. To maintain these properties and to ensure the completeness of the requirement specification document, much effort has been directed towards developing appropriate tools for automating the requirement phase.

(2) System Functions Specification

The design process begins after the requirements phase has been completed. Design consists of an orderly decomposition of the system functional components into sub-functions. The first level of decomposition is usually carried out without specifying the architectural alternatives for implementing the set of sub-functions of the multiprocessor system. Thus this phase of the design process involves defining the functional components of the system which collectively satisfy the stated requirements.

(3) System Architecture Selection

At the architectural level, the designer is concerned with the overall structure of the system. This involves the components (processors, memories, etc.) and their interconnections in such

configurations that are likely to satisfy the specifications (expressed in terms of functional components). Each of the architectural components has certain quantitative attributes which determine its performance (eg. memory size, processor speed, bus data transfer rates, etc.).

(4) System Model

The architectural description of the system can be abstracted at this level into a system model which includes description of the algorithmic behaviour of certain functions as well as a description of the structure of each architectural component and the interconnection of the various components. At this level, the relationship and trade-off between the hardware and the software of the system can be analysed and understood.

(5) Register Transfer Level

At the Register Transfer level, the designer is concerned wih realizing the functions described in the system model by sequences of operations. These operations are usually specified as transfers of information between the different faciilities established in the architectural and system model steps. Thus if the multiprocessor system is considered as a large finite state machine, then the purpose of the register transfer design level is to establish the various states as well as the particular actions to be taken when the system is in a given state.

(6) Logic Design Level

At the Logic Design level, the designer is concerned with the mapping of the microoperations and the control structure, defined in the previous step, into physical hardware elements. This step requires a detailed knowledge of the technology in which the design is to be implemented (eg. IC components on a PCB, LSI or VLSI technology, etc.).

The survey presented in this report groups the design tools into four categories, depending on the design level at which a given tool is utilized. These categories are:

(i) Tools for specification of system (multiprocessor) requirements. Such tools are useful for design level (1) described above. A survey of such tools is provided in Section 2.1 (Requirements Specification Tools).

(ii) Tools for definition of system functions to satisfy the requirements. Such tools are useful for design level (2) described above and are discussed in Section 2.2 (Tools for Definition of Functional Components).

(iii) Tools for simulating system architecture and system model. These tools are useful for design levels (3) and (4) described above and are discussed in Section 2.3 (Architecture and System Model Simulation Tools).

(iv) Tools for modelling and simulation of the system model at the register transfer level. These tools are also known as hardware

description and simulation languages. They are used at design level (5) described above. Section 2.4 presents a survey of these tools (Register Transfer Level Simulation Tools).

Our survey does not cover the logic design level tools. As indicated earlier, the structure and utility of these tools are closely related to the technology adopted and are beyond the scope of this study.

The survey of multiprocessor CAE tools presented in Section 2 of this report has been prepared with reference to the design levels described above and illustrated in Figure 1.1. It should be noted, however, that such categorization of the design tools does not represent a universally acceptable standard, nor can the lines defining the boundaries of each design level be sharply drawn. In fact, several of the design tools surveyed may cross the boundaries between design levels and may also serve in the design process at two or more successive levels.

Finally, the survey presented in this report is not claimed to be an exhaustive treatment of all existing and planned multiprocessor CAE design tools. More accurately, the survey may be viewed as an attempt to present current approaches and techniques in the field of CAE tools for multiprocessor design. The authors are confident that the systems included in the survey are representative of such approaches and techniques.

2.0 Survey of Existing Multiprocessor CAE Design Tools

2.1 Requirements Specifications Tools

The term "requirements" is used to describe any demand upon the system. Before the system is actually designed and implemented, all the requirements should be defined and be consistent, unambiguous and non-redundant.

Traditionally, requirements specification documents were written in English. However, this choice of language led to two problems:

1. the inherent ambiguity of English led to major disagreements between the requirements writers and system designers, and
2. the requirements did not lend themselves to machine processing.

Automated requirements tools offer solutions to these two problems. First, these tools could be used as processors for formal requirements languages with compact vocabulary and semantics. Second, the tools produce documents that could be processed by other software tools.

Two tools are currently in use which can be regarded as purely requirements tools. These are:

1. the Requirements Engineering and Validation System (REVS), authored by TRW Defense and Space Systems [ALF077], [DAVI77], and
2. the Requirements Processing System (RPS), from GTE Laboratories [DAVI79a], [DAVI79b].

Many other tools reported in the literature have been referred to as requirements specification tools even though they tend to support the

design process through decomposition of the system into functional components. Such tools should not be regarded as pure requirements specification tools, ie. they do not fall into the same category of the two tools listed above. A survey of these tools is provided under the Functional Component Definition category of tools (Section 2.2).

In general, REVS and RPS consist of a table-driven compiler that allows the user to define the features of the system in a language specifically tailored for his application area. Thus, while the syntax of the language is defined, the user has the freedom of introducing his own vocabulary and semantics within the given syntax rules.

Using REVS or RPS as tools for requirements specification provides the following advantages:

1. Multiple Authorship

   Many groups of people, each working independently, may write their sections of the requirements. The requirements specification tool merges all the independently written sections into one coherent, cohesive and fully integrated system description.

2. Document Formatting

   The requirements specification tool can serve as an automated formatter that is capable of producing formatted text, table of contents, and cross-reference indexes.

3. Consistency Checking

   The requirements specification tool reports any violation in the specification in the areas of incompleteness, inconsistency, ambiguity and redundancy. The generated output will be in a machine readable

format which lends itself to further processing, if needed, during later design phases.

2.2 Tools for Definition of Functional Components

Design occurs after the requirements phase has been completed. The first level of the design process consists of an orderly definition of the main functional components of the system which satisfy the requirements. Next, these functional components are further decomposed into smaller sub-functions, and the process continues until a system architecture emerges in which a hardware/software model can be abstracted and the sub-functions can be mapped into elements in the model.

Several tools are currently available for automating the process of defining the functional components of the system and for decomposing these components into smaller, less complex components.

Initially, each component is defined in terms of its external interfaces to other components and the specific feature (or features) of the requirements it is supposed to satisfy or perform. Once the external behaviour of all components is completely specified, the decomposition process can begin and can be applied to each component separately and independently. Each component can be decomposed into subcomponents; each subcomponent is described, in turn, in terms of its external behaviour (interfaces). As long as no changes are introduced by the decomposition process to affect the external behaviour of the component being decomposed, the refinement process can be performed on the components separately.

A decomposition process along the lines described above leads to a "top-down" design approach. Practical experience with complex systems indicates that such an approach is necessary for reducing the development effort and keeping the design process within controllable bounds.

Several tools have been developed in recent years for automating the definition and decomposition of system functional components. The most widely known of these tools are:

- the Problem Statement Language/Problem Statement Analysis (PSL/PSA) System from the University of Michigan [TEIC77]
- System Analysis and Design Techniques from SofTech [ROSS77]
- AUTO-IDEF from the Computer Corporation of America [LIPK80]
- Input-Output Requirement Language (IORL) from Teledyne [EBER80]
- the System Design Processing SPD from GTE [ROMA79].

The above set of tools differ in the syntax of the language used and the details for the functional decomposition utility offered to the designer. The tools, however, tend to offer similar services to the designer in the following aspects:

- automated documentation of the functional components of the system. Each component is defined in terms of its external interfaces to other components.
- validation check of the functional decomposition process, ie. ensuring that a component may not accept a message unless another component sends it. Also ensuring that the decomposition of a component into

subcomponents does not violate the external interfaces associated with the definition of the component.

- ensuring that all the features defined in the requirements specification are satisfied by at least one component.

2.3 Architecture and System Model Simulation Tools

The next step following the decomposition process of the functional components of the system is to select an overall structure with components such as memories, processors and I/O devices which can best satisfy the requirements explicitly stated in the description of the functional components. The selection of the most suitable structure is based on cost/performance and other related factors (eg. availability). The cost/performance factor is derived for each architecture as a function of the quantitative attributes of the structural components (eg. memory size, CPU speed, word length, bus bandwidth, etc.).

Existing CAE tools at the architectural level tend, in general, towards assisting the designer in evaluating a number of architectures (selected by the designer based on past experience) and selecting the best architecture based on the evaluation criteria.

The evaluation can be conducted by first formulating a model using the tool. This is known as model definition. The tool checks the consistency of the model definition and then provides the designer with a simulation environment in which the performance aspects of the modelled architecture can be monitored.

The tools available at the architectural design level can be classified into two types:

1. General Purpose Simulation Languages:

   Examples of these languages are GPSS [GENE73], SSH [ROHM77] and SIMULA [DAHL76]. A model must be constructed for the architecture and defined using the syntax of the language. Following compilation, the defined model can be run using simulated input data (transactions). The results of the simulation runs contain information related to high level resource utilization aspects such as throughput, delay, bus congestion, etc. Once analysis at this level is completed, the design is refined into more concrete hardware and software components which can be studied using the second type of tools described below.

2. Special Purpose Simulation Languages

   These simulation languages have been developed to serve as simulation tools for single processors and multiprocessor systems. The general approach followed, as indicated in the three example tools surveyed in the remainder of this section, is to provide the designer with two languages: one language is used to describe the intended behaviour of the system (behaviour description language) and a second language is used to describe the actual implementation details (structure description language). The model of the system being designed consists of the compiled outputs of the two languages for the source code which represents the model behavioural and structural definitions. The two compiled outputs are linked and then used as part of a runtime simulation package controlled by the user.

The remainder of this section presents three tools which have been developed for the Architectural and System Model design level. The tools follow the conceptual approach described above.

2.3.1 AIDE - ArchItecture Design Environment

AIDE is a modelling and simulation system designed to support the development of computer architectures. The system is optimized for the modelling and evaluation of computer hardware architectures [ELLE81]. The area of applicability of AIDE ranges from the system architecture level to the micro-architecture (internal hardware organization) level.

The AIDE system consists of three components (shown in Figure 2.1):

(1) the Language Environment,

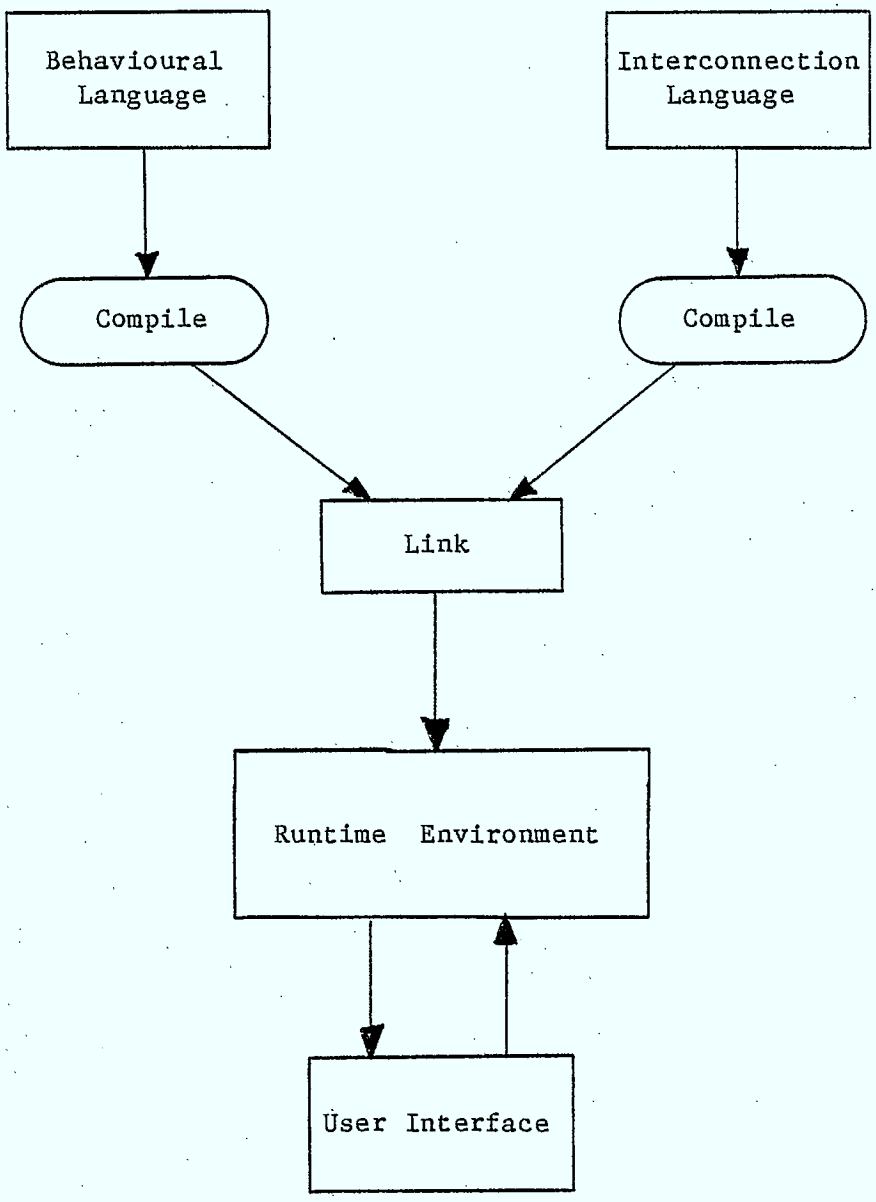
(2) the Runtime Environment, and

(3) the User Interface.

The language environment allows the user to define models for the computer system in the form of behavioural modules; each module contains a set of functions connected to the outside world by a set of inputs and outputs. The path of interaction between the modules is defined by an interconnection structure which links the outputs of modules to the inputs of other modules. Thus an AIDE model requires the use of two languages: a behaviour description language and an interconnection description language.

Figure 2.1

AIDE Components



The runtime support package provides the mechanisms needed to allow concurrent behavioural execution, manage system resources and support performance evaluation.

Simulation control is linked to the user interface, where the user interactively issues execution commands. These commands allow the user to examine aspects like performance analysis in addition to rudimentary tasks (stop, go, store results in a file for later processing, etc.).

A description of the above three components is detailed in the following.

(1) The Language Environment

As mentioned earlier, the AIDE model requires the use of two languages: a Behaviour Description Language (BDL) and an Interconnection Description Language (IDL). The BDL in AIDE is an extension of the C programming language in two areas. The first involves "modularization" of model descriptions. The second involves the support of additional data types and constructs to model concurrent and asynchronous functions. The description of each module begins with the type name of the module, followed by declarations of all inputs and all outputs. These declarations act as interface ports to other behaviour modules.

As an option, a module may contain "state" variables which are global to all the code within the module, but are not accessible from outside the module. Examples of these state variables include memory arrays, registers and state variables.

The module body follows the input/output port and state declarations. It contains the code which specifies the behaviour of the module in terms of a set of constituent processes. Processes within the same module share a common scope and can interact with each other directly. For inter-module process interaction, a process is always attached to one or more input ports of the module in which it is defined. The interconnection language links ports together and defines the interaction path between the respective processes.

The AIDE Interconnection Design Language (IDL) allows the description of single entities called "macros". A macro is externally identical to a behaviour module but contains no behaviour code. Instead, a macro acts as a shell that identifies a logically unique subsystem which consists of a number of modules. Once defined, a macro may be used either as a complete system or as a "module" in some other system.

Typically, a macro is first compiled as a system and is evaluated separately. After testing, the macro may be recompiled as a component for subsequent use in layered systems. The macro approach has several advantages:

i. Components containing complex functionality can be built and tested piecemeal. Design synthesis is facilitated through this button-up approach.

ii. Each component may be placed in a library containing other modules or macros and recalled for later design. As the library grows, the work involved for new developments shrinks.

iii. The macro approach also aids the top-down design process by providing consistent interfaces within a model undergoing refinement.

(2) The Runtime Support Package

The organization of the runtime support package in AIDE is shown in Figure 2.2. The model developed in the language environment is first compiled and then linked with the runtime support package. The resultant executable load module is called a "simulation model". The runtime simulation support for the model consists of four components:

i. Scheduler: The AIDE Scheduler fields events generated during simulations, schedules and executes them at the proper time. The events are scheduled with respect to a global system clock. Most of these events are generated by the model. However, the scheduler also handles special control events initiated by the user.

ii. Memory Monitor: This component contains software memory management routines so that a user need only declare memory in his model using the state variable declaration. The simulated memory manager uses a software paging scheme to create a file for each paged memory declared in the user's model.

iii. Performance Monitor: Users are allowed to construct commands meaningful to their application from several AIDE native commands. User commands can create triggers during simulation to detect conditions on monitored variables. When

one in a list of triggers fires, a specified command list is executed. The performance monitor also provides a mechanism for recording and manipulating performance information.

iv. The Command Interpreter: The command interpreter communicates with the user interface by sending and receiving messages. It thus provides a central point of control and access to the simulation model.

(3) The User Interface

The main purpose of the user interface is to provide a friendly environment for the user to control and evaluate a simulation. It communicates with the command interpreter when requesting some runtime action. User commands include sequencing commands (go, halt, etc.), query commands regarding the status of the simulation, and other commands to load and display model variables and simulated memory locations.

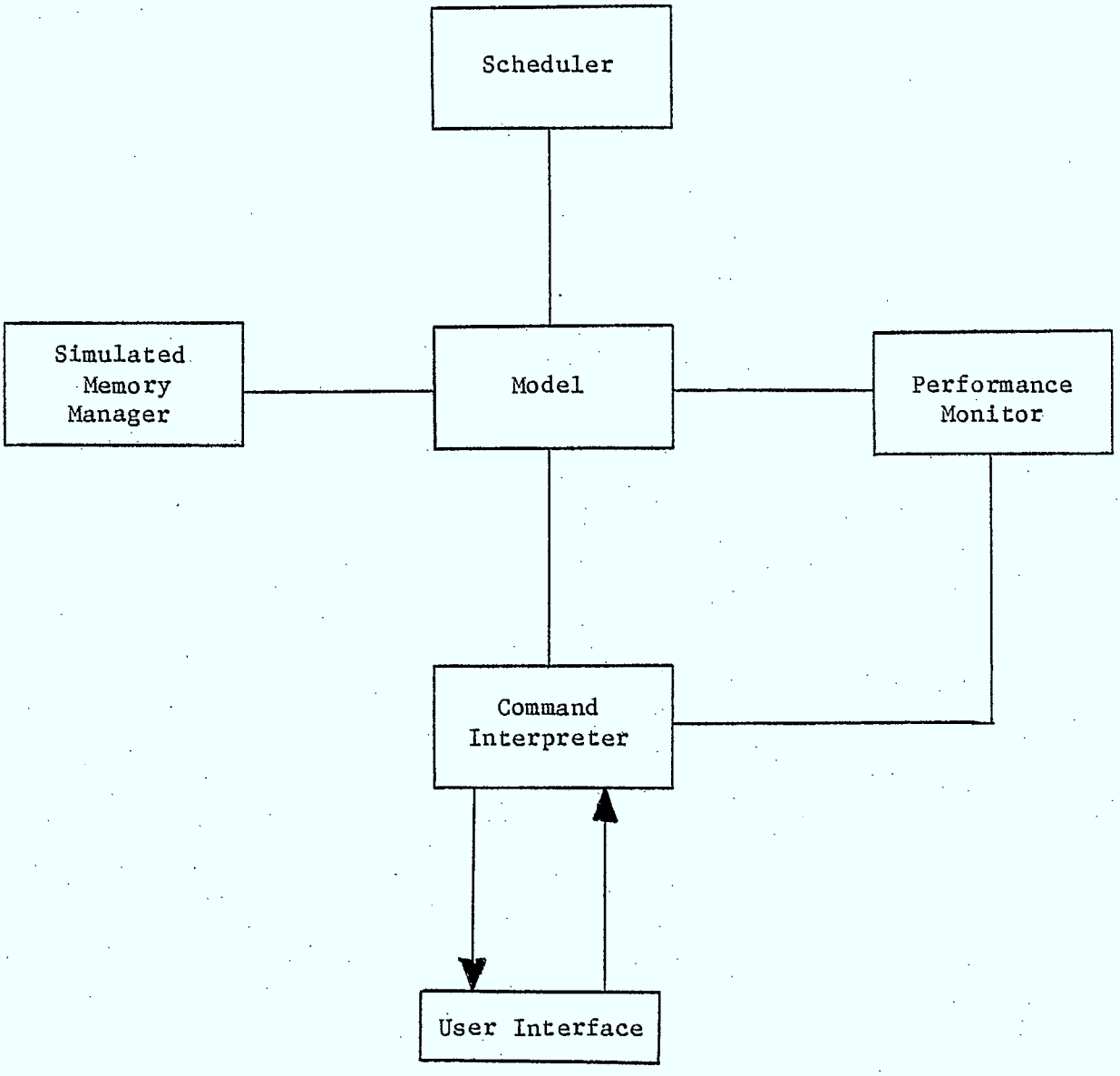
2.3.2 N.mPc Design Environment

N.mPc is an interactive environment for the design and evaluation of multiprocessor systems ([PARK79a], [ROSE79], [PARK79b], [ORDY79]), developed and implemented at Case Western Reserve University. It contains six separate components which work together to produce function, register transfer level simulations of multiple processor, heterogeneous target systems.

Figure 2.3 presents a simplified system block diagram of the N.mPc system components. The "meta assembler" allows the designer to specify the details of the target instruction set in a format which is

Figure 2.2

AIDE Runtime Environment



machine independent. The "linking loader" resolves the machine dependent aspects of metamicro assembler description output and allocates the resulting code to physical memory according to user selected memory allocation strategy. The linking loader generates a "simulated memory processor".

The ISP Compiler is used to translate processor and interconnection element descriptions, defined using a hardware description language, into executable modules. These modules are linked with the target system topology description by the "Ecologist" component which generates the "simulation model program". The simulation model program, also known as the ´kernel´, runs under the control of a "Runtime Package". The Runtime Package consists of a Command Interpreter, the Kernel and the Simulated Memory Manager. The Kernel and Command Interpreter provide the user with interactive control and monitoring of simulations. The Simulated Memory Manager supervises memory content to optimize the performance of the simulation.

A brief description of the six components of the N.mPc system is provided in the following:

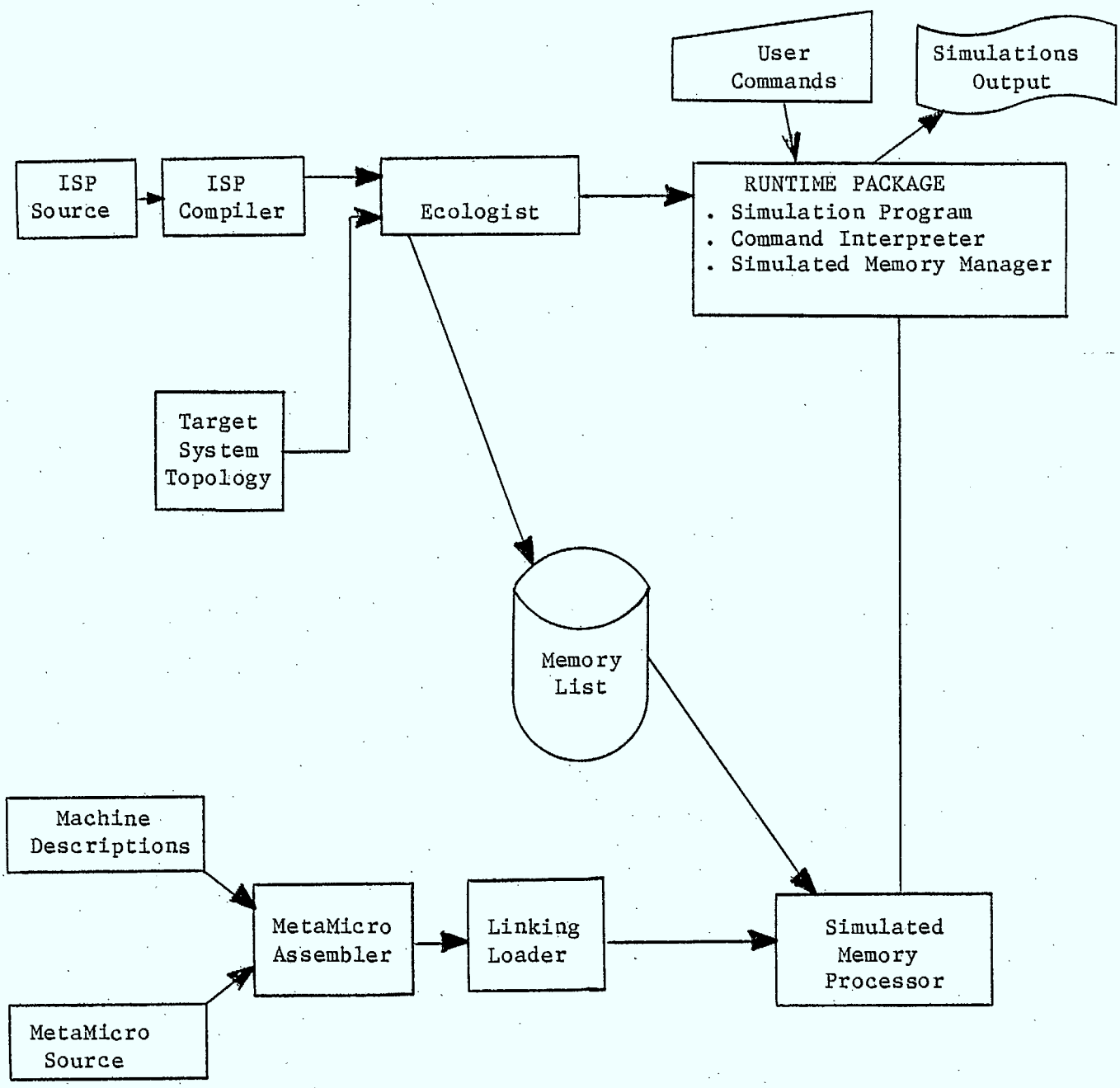
(1) MetaMicro Assembler

The assembler contains a set of facilities for both vertically and horizontally organized target architectures. The facilities include the following main sections:

- A declaration section to describe the structure and semantics of the target machine.

Figure 2.3

Main Components of the N.mPc System



- An instruction section to assemble the source program instructions.
- A register transfer notation section to provide a source instruction format using register transfer notation and "opcode-operand" notation.
- An allocation control section to provide information to the linking loader concerning absolute address and contiguous code requirements.

In addition, the assembler contains facilities to handle variable length instructions, predefined source text inclusion, external/internal interface functions and illegal opcode checking.

(2) Linking Loader

The linking loader accepts the output of the metamicro assembler along wih a "command" program which defines the physical memory constraints and the semantics of the target machine. These semantics include an instruction and format declaration section, the addressing modes of the target processor, and a set of rules for allocating physical memory spaces according to one of four user specified algorithms.

(3) ISP Compiler

A register transfer language, ISP, is used to describe the hardware modules along with a compiler which converts ISP module description into executable code. ISP allows the descriptions of multiple processor/module architectures through the use of "PORT" constructs which interface the module to the external environment. Synchronization between the modules can be achieved by a 'WAIT'

construct. Events within a processor may be separated in time by the use of a "DELAY" construct which specifies the execution or delay time associated with the previous register transfer statement.

(3) Ecologist

The Ecologist generates an executable program for the simulation of the desired target architecture. It accepts as input the compiled ISP modules for each of the hardware components of the target system and connects the specified hardware ports. To do so, the ecologist must obtain as input the specified topology of the target architecture. The topology is described in the form of a series of declarations:

- Signal Declarations: define control lines and data buses
- Processor Declarations: create an instance of a processor which can be referred to by name at runtime
- Time Delay Declarations: define the time unit to be used with the DELAY statement
- Connection Declarations: connect the PORTS of an ISP processor to signals defined in the signal declarations
- Initial Declarations: bind an ISP processor memory to either a linking loader output file or to a UNIX (operating system) file or device.

(5) Simulated Memory Processor

The Simulated Memory Processor prepares memory files for use by the Runtime Package. All linking loader output files needed by the simulation are converted from the packed format produced by the linking loader to the segmented or paged format required by the simulation program. The Simulated Memory Processor also creates a memory symbol table file containing the names of the memory files available to the simulation.

(6) Runtime Package

The Runtime Package consists of three components:

- the simulation model (kernel) generated by the Ecologist
- the Simulated Memory Manager (with input from the simulated memory processor)
- a runtime command interpreter connected to the user interface.

The simulation kernel performs ISP process scheduling and data manipulation functions required for the execution of the simulation. The kernel also manages monitor functions used for automatic collection of simulation data.

Simulated memories are handled by the Simulated Memory Manager. Up to 128K bytes of simulated memory may reside in the main memory of the supporting computer (PDP11).

The runtime command interpreter handles the interface between the simulation user and the simulation tool itself. The interpreter accepts commands from the user to examine or modify simulation states, control the execution, set execution breakpoints and collect data from a running simulation.

2.3.3 SABLE: A Tool for Generating Structured Multi-level Simulations

SABLE's approach is similar to that of other simulation languages at the architectural level in that it uses a structure specifications language to specify the nesting and interconnectivity of components, and a general purpose algorithmic language to describe each component's behaviour. SABLE then analyzes the structure of the system and connects the appropriate behavioural descriptions accordingly [HILL79a], [HILL79b].

The concept of components operating at various data levels is central to the top-down design methodology adopted in SABLE, which encourages simulation prior to detailed design. Like other design automation tools, SABLE models a computer system as a collection of components interconnected by nets. Each component is an instance of a particular component type, abbreviated comptype, such as "nand-gate" or "controller". The structure of the component is described using a language called ADLIB.

When developing a new system, the SABLE user can start at the topmost nesting level of his design and decompose it into a few interconnected components. He then specifies the desired behaviour of each component by writing an ADLIB comptype. He runs SABLE and uses the resulting simulation to evaluate overall configuration and performance. Gradually, the components are decomposed into smaller functional blocks, which may now include both hardware and software units. The behaviour of each is again specified in ADLIB. The decomposition and refinement of components is recursive as described earlier.

The design at each point can be evaluated by checking if the simulated behaviour of the refined design matches that of the higher level component that it was supposed to implement. Errors in the refined design can be easily identified if any mismatch is found.

SABLE has been implemented on a DECSYSTEM-20 using about 5000 lines of PASCAL. The support software consists of three parts: the ADLIB precompiler, the runtime support code, and the SABLE module itself, which also contains a simple SDL compiler. In order to use the system, a user must provide an ADLIB and an SDL source file, though the SDL file may be generated with an interactive graphic structure editor.

The user first runs the ADLIB precompiler, which parses the program and produces several files. These include a simple database that reflects all the external attributes of each comptype, ie. its name, nets and parameters. If no ADLIB syntax or semantic errors were detected, the precompiler generates a valid PASCAL program using the ADLIB source and the runtime support code. The user then runs SABLE, which compiles his SDL source and generates a topology file and a parameter file. The standard PASCAL compiler is then used to compile the reformatted ADLIB program, and simulation begins. During the start of simulation, part of the runtime support code uses the topology file to interconnect and activate the desired selection of components.

2.4 Register Transfer Level Simulation Tools

The Register Transfer Level is concerned with realizing the functional specifications by sequences of operations. If we consider a digital system as a large finite state machine, then the purpose of register-transfer level design is to establish the various states in which the system may find itself, as well as the particular actions to be taken when the system is in a given state.

The register level is one at which most computer hardware description languages are used. As in the architectural and system model level, hardware design languages tend to fall into one of two categories: Behaviour Description Languages and Structure Description Languages.

Hardware design languages in both categories have been used extensively over the past decade and are well documented in the open literature. In view of the availability of excellent surveys covering this level of design, the discussion of this section will not be extended beyond the references to relevant publications.

An excellent survey of all major computer hardware description languages can be found in [SU74]. A bibliography on the subject can be found in [VANC76], [VANC77], [VANC78]. Finally, a survey of the applications of different hardware design languages can be found in [VANC79].

Several multiprocessor systems development tools that are available commercially fall into the register transfer simulation category of tools. Typically these tools are developed for specific microprocessor (eg. Intel 8086 or Zilog Z80/CPM) and allow the designer to simulate the target code

for the processor on a host machine (eg. PDP11/45) to check its correctness and rectify any errors. As well, tools such as in-circuit emulators are usually provided to allow the user to perform on-line debugging and testing of the code prior to the field testing stage.

3.0 Summary and Conclusions

A survey of CAE tools for multiprocessor design has been presented. The survey identified six specification and design phases for the purpose of identifying the utility and applications of the various available tools. These phases are:

(1) the Requirements Specification phase,

(2) the Functional Components Definition phase,

(3) the Architectural Design phase,

(4) the System Model (Hardware/Software trade-off) phase,

(5) the Processing Element Partitioning (Register Transfer Level) phase, and

(6) the Logic Design (Hardware) phase.

The survey presented covered existing tools for the first five design and specification phases. It was pointed out that the lowest level (Logic Design) is related to the technology used (eg. PCB/IC, VLSI, etc.) and is beyond the scope of this survey.

Many of the tools surveyed do not necessarily fall exclusively within the boundaries of one design level. In fact, tools that are based on a top-down design methodology tend to span the boundaries of several design levels and can thus be used iteratively throughout several design phases.

The general concept underlying the top-down design approach can be summarized as follows:

- Following the requirements specification phase, the main functional components of the system are defined. Each component is described in terms of its external interfaces and the basic functions it performs.

- Each component can then be treated separately and divided into subcomponents. Collectively, these subcomponents must maintain the same external interfaces of the parent component. Again, each subcomponent is described in terms of its external interfaces.

- The process of decomposition can be applied iteratively to the subcomponents until they are refined into the simplest possible (elementary) components.

- A validity check is applied at each decomposition step to ensure its completeness among the subcomponents (the output/input of each subcomponent is either an input/output to another subcomponent or is an output/input of the parent component). The validity check is also applied to ensure the consistency of the decomposition process.

The above four steps constitute the design specification path (the top-down direction). The dual process, called the design synthesis, takes place in the opposite direction (bottom-up) and follows the design specification process. Design synthesis starts with the most refined (elementary) components and assembles the pieces to form high level components. The synthesis continues until the main components are formed.

The survey presented here indicated the availability of many design simulation tools which satisfy different design needs, depending on the design level (or levels) for which it is developed. Unfortunately, no one simulator is useful throughout all specification and design phases. This

multiple simulator approach outlined in the survey has two advantages and several disadvantages. The advantages are:

1. Each simulation can be written in a language tuned for one particular level, and

2. Each simulation tool can optimize its runtime organization for one particular task.

The disadvantages include the following:

1. The design effort is multiplied by the necessity of learning several simulator systems and writing a design in each.

2. The possibility of error is increased as more human manipulation is involved.

3. As the design becomes increasingly fragmented, it becomes impossible to simulate an entire multiprocessor system at a low level of abstraction. Therefore only small fragments can be simulated at any one time.

4. Each fragment needs to be driven by a supply of realistic data and its output must be interpreted. This may make the sofware written to serve these needs extremely costly.

Several tools have been developed to overcome the above difficulties and provide the designer with a uniform simulation approach starting at the architecture design level and going down to the register transfer simulation level. Examples of these tools are the AIDE package (Bell Laboratories) and SABLE (Stanford University). The utility of these tools can be improved substantially by augmenting them with a high level

specification package which allows the designer to describe the functional components of the system being designed and to interface this high level description to existing tools at the architectural level. In addition, two design aspects must be addressed in augmenting existing tools:

1. Redundancy and fault-tolerance characteristics analysis must be provided at the architectural levels if the tools are to be useful in the design of spacecraft (or avionics) multiprocessor systems.

2. The high level specification language must contain mechanisms for system verification. As the design process continues, these mechanisms will evolve naturally towards validation and verification of the software.

The issue of augmenting existing tools to generate an integrated set of multiprocessor design and simulation tools is addressed in a separate study report (Report #3, referenced in the preface of this report).

4.0 References

[ALFO77] M. Alford, "A Requirement Engineering Methodology for Real-Time Processing Requirements", IEEE Transactions on Software Engineering, SE-3, pp. 60-69 (1977).

[DAHL76] O.J. Dahl and K. Nyggard, "SIMULA: An Algol based Simulation Language", Communications of the the ACM, Vol. 9, September 1976.

[DAVI77] C. Davis and C. Vick, "The Software Development System", IEEE Transactions on Software Engineering, SE-3, pp. 69-84, (1977).

[DAVI79a] A. Davis and T. Rauscher, "Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications", IEEE Conference on Specifications of Reliable Software, Cambridge, Massachusetts, pp. 15-35, (1979).

[DAVI79b] A. Davis, et al, "RLP - An Automated Tool for the Automatic Processing of Requirements", Proceedings of COMPSAC 79, Chicago, Illinois, pp. 289-299, (1979).

[EBER80] C. Eberhard, presentation on "IORL" and panel discussion "Software Requirements Engineering", NCC 80, May 1980, Anaheim, California.

[ELLE81] D.J. Ellenberger and Y.W. Ng, "AIDE - A tool for Computer Architecture Design", 18th Design Automation Conference (IEEE), pp. 796-803, 1981.

[GENE73] "General Purpose Simulation System/360 User's Manual", Form H20-0326, IBM Corp., White Plains, N.Y., 1973.

[HILL79a] D. Hill and W. VanCleemput, "SABLE: A Tool for Generating Structured Multi-level Simulation", Design Automation Conference, 1979, pp. 272-279.

[HILL79b] D. Hill, "ADLIB - SABLE User's Guide", Computer Systems Lab Tech Report, Stanford University, Stanford, 1979.

[LIPK80] S. Lipka, "Software Requirements Engineering: A Tool Developers View on AUTOIDEF", NCC 80, May 1980, Anaheim, California.

[ORDY79] G.M. Ordy and F.I. Parke, "An Evaluation of the N.mPc. Design System", Proceedings 16th Design Automation Conference, pp. 537-541, June 1979.

[PARK79a] F.J. Park, "An Introduction to the N.mPc. Design Environment", Design Automation Conference, pp. 513-519, June 1979.

[PARK79b] F.J. Park et al, "The N.mPc. Runtime Environment", Proceedings 16th Design Automation Conference, June 1979, pp. 529-536.

[ROHM77] J. Rohmer, "SSH Simulator for Hierarchical Systems", Proceedings 10th Annual Simulation Symposium, pp. 109-127, 1977.

[ROMA79] J. Romanos, "The Software Design Processor", Proceedings of COMPSAC 79, Chicago, Illinois, pp. 380-383 (1979).

[ROSE79] C.W. Rose, et al, "The N.mPc. System Description Facility", Proceedings 16th Design Automation Conference, June 1979, pp. 520-528.

[ROSS77] D. Ross and K.E. Shoman, Jr., "Structured Analysis for Requirements Definition", IEEE Transactions on Software Engineering, SE-3, pp. 6-15 (1977).

[SU74] S. Su, "A Survey of Computer Hardware Description Languages in the U.S.A.", IEEE Computer, Vol. 7, No. 12, pp. 45-51, December 1974.

[TEIC77] D. Teichroew and E.A. Hershey, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", IEEE Transactions on Software Engineering, SE-3, pp. 41-48 (1977).

[VANC76] W.M. vanCleemput, "Computer-Aided Design of Digital Systems: a Bibliography", Woodland Hills, Cal.: Computer Science Press, 1976.

[VANC77a] W.M. vanCleemput, "Computer-Aided Design of Digital Systems: a Bibliography, volume 2: 1975-76", Woodland Hills, Cal.: Computer Science Press, 1977.

[VANC77b] W.M. vanCleemput, "A Hierarchical Language for the Structural Description of Digital Systems", Proceedings 14th Design Automation Conference, New Orleans, June 1977, pp. 378-385.

[VANC78] W.M. vanCleemput, "Computer-Aided Design of Digital Systems: a Bibliography, volume 3: 1976-1977", Woodland Hills, Cal.: Computer Science Press, 1978.

intellitech
Intellitech Canada Ltd.
352 MacLaren Street,
Ottawa, Ontario.
K2P 0M6
(613)235-5126